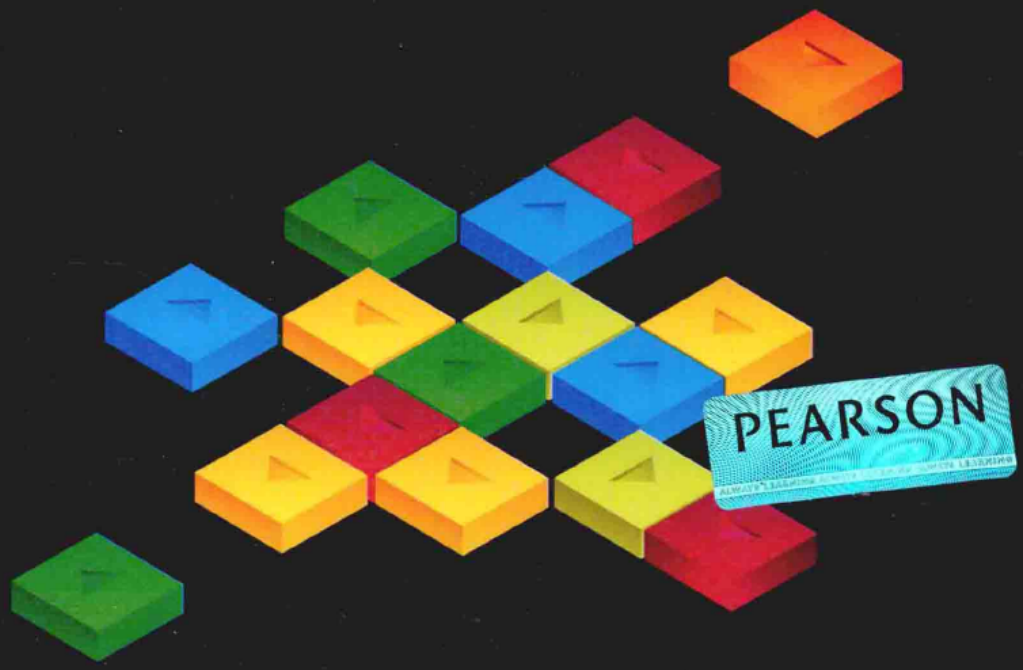
世界著名计算机教材精选

PEARSON

Java软件结构与数据结构（第4版）

John Lewis Joseph Chase 著

金 名 等译



JAVA SOFTWARE STRUCTURES

DESIGNING AND USING DATA STRUCTURES

Fourth Edition

清华大学出版社

JAVA SOFTWARE STRUCTURES
DESIGNING AND USING DATA STRUCTURES Fourth Edition

Java软件结构与数据结构（第4版）

本书由著名作者John Lewis编写，本书始终以良好软件工程实践为准则，以Java为语言，介绍数据结构与算法的相关知识和具体实现，强调完美软件设计技巧的重要性，并通过大量的示例，支持和强化数据结构与算法的学习。

因此，本书不仅是数据结构与算法课程的理想教材，也可以作为Java程序设计课程的相关教材。

世界著名计算机教材精选

- C++面向对象程序设计（第4版）
- Java程序设计：一种跨学科的方法
- Java程序设计与问题解决（第6版）
- Java面向对象程序设计（第2版）
- Java软件结构与数据结构（第4版）
- TCP/IP协议原理与应用（第4版）
- TCP/IP协议族（第4版）
- Web数据挖掘（第2版）
- 编译器构造（Java语言版）
- 编译器设计基础
- 标准C程序设计（第5版）
- 操作系统原理
- 操作系统原理、设计与应用
- 程序设计基础（第5版）
- 程序设计语言概念（第9版）
- 分布式数据库系统原理（第3版）
- 分布式系统原理与范型（第2版）
- 计算机安全导论
- 计算机图形学（OpenGL版）（第3版）
- 计算机图形学与几何造型导论
- 计算机网络（第5版）
- 计算机网络数学基础
- 计算机系统设计原理
- 计算机组成和设计硬件／软件接口（第2版）
- 计算机组织与体系结构：性能设计（第7版）
- 计算几何：算法与应用（第3版）
- 计算理论基础（第2版）
- 计算群体智能基础
- 计算智能导论（第2版）
- 局域网与互联网原理与应用
- 离散数学与组合数学（第5版）
- 密码学与网络安全（第2版）
- 面向对象软件工程：使用UML、模式与Java（第3版）
- 面向对象设计UML实践(第2版)
- 面向对象系统分析与设计（第2版）
- 面向计算科学与工程的MATLAB编程
- 人工智能：一种现代的方法（第3版）
- 数据结构基础（C++语言版）（第2版）
- 数据结构基础（C语言版）（第2版）
- 数据结构与面向对象程序设计（C++版）（第4版）
- 数据结构与算法分析（C++语言描述）（第2版）
- 数据结构与算法分析（Java语言描述）（第2版）
- 数据结构与问题求解（Java语言版）（第4版）
- 数据库管理系统原理与设计（第3版）
- 数据库系统基础（第6版）
- 数据挖掘教程
- 数值分析与科学计算
- 数字图像处理：原理与应用
- 算法设计
- 算法与并行计算
- 算法与数据结构
- 图像处理、分析与机器视觉（第3版）
- 图像处理基础（第2版）
- 网络安全基础：应用与标准（第5版）
- 问题求解与程序设计（C++语言版）（第6版）
- 无线通信与网络（第2版）
- 系统分析与设计基础（第5版）
- 语义Web技术基础

ISBN 978-7-302-34670-8
9 787302 346708 >
定价：79.00元

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

PEARSON
http://www.pearsoned.com

世界著名计算机教材精选

Java 软件结构与数据结构

（第 4 版）

John Lewis
Joseph Chase 著
金 名 等译

清 华 大 学 出 版 社

Simplified Chinese edition copyright ©**2014** by **PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.**
Original English language title from Proprietor's edition of the Work.
Original English language title: Java Software Structures: Designing and Using Data Structures, Fourth Edition **by** John Lewis, Joseph Chase, Copyright © **2014**
EISBN:978-0-13-325012-1
All Rights Reserved.
Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.
This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).
本书中文简体翻译版由 Pearson Education（培生教育出版集团）授权给清华大学出版社在中国境内（不包括中国香港、澳门特别行政区）出版发行。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

Java 软件结构与数据结构：第 4 版/（美）刘易斯（Lewis, J.），（美）蔡斯（Chase, J.）著；金名等译. --北京：清华大学出版社，2014
书名原文：Java Software Structures: Designing and Using Data Structures, 4e
世界著名计算机教材精选
ISBN 978-7-302-34670-8

Ⅰ. ①J…　Ⅱ. ①刘…　②蔡…　③金…　Ⅲ. ①Java 语言-程序设计-教材　②数据结构-教材　Ⅳ. ①TP312　③TP311.12

中国版本图书馆 CIP 数据核字（2013）第 290834 号

责任编辑： 龙启铭
封面设计： 常雪影
责任校对： 焦丽丽
责任印制： 杨　艳

出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn,010-62795954
印 刷 者： 北京富博印刷有限公司
装 订 者： 北京市密云县京文制本装订厂
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 31.5　　**字　　数：** 764 千字
版　　次： 2014 年 4 月第 1 版　　**印　　次：** 2014 年 4 月第 1 次印刷
印　　数： 1～2000
定　　价： 79.00 元

产品编号：054403-01

译 者 序

这是这本畅销书的最新版——第 4 版了，著名作者 John Lewis 与 Joseph Chase 在前几版的成功基础上，根据使用本书做教材的教师和学生的反馈，在新版中进行了大量改进，以更好地适应教学需要。最重要的修改包括这样几个方面：

- 修订了有关集合的章节，对 Java API 是如何支持集合的给出了更全面的阐述。
- 在每章的后面增加了术语和定义的汇总。
- 把迭代器单独成一章，并对它进行了扩展介绍。
- 增加了新的代码注解。
- 增加了新的常见错误插图。
- 增加了新的设计要点插图。
- 增加了一个新附录，介绍图形绘制、图形用户界面开发以及常规表达式。
- 对全书进行了修订和更新，以提高介绍，解决问题。

总之，这些修改都是建立在使用以前版本教学的基础上，为教师提供更多的机会和更好的灵活性来使用本书。

在翻译过程中，我们力求忠实原文，但个别地方，我们认为作者的表述不是很清楚，或者认为直译成中文容易使读者产生困惑，我们在不改变原意的基础上，对个别表述方式进行了修改，或添加译注说明来进一步阐明作者的本意。

本书由金名、张长富主译，陈宗斌、陈河南、陈洁荣、陈征、丁梦桐、戴君、贺军、胡嵬、黄达明、马宏华、王玉玲、谢淘、张海涛、郑朦、郑旭、李翔鹰、邱海艳、徐晓蕾、魏世亮、吕晓晴、李晓宁、位凯茜、韩珺、刘妍、郑炳晴、宋丹、傅强、宋如杰、蔡江林、戴锋、蔡永久、龚宇华、何正雄、黄定光、来阳、李刚生、李韬、欧婷、乔健、苏高、孙朝辉、孙丽、徐茜、许瑛琪、叶守运、易陈丽、叶淑英、易小丽、喻四容、易志东、殷小俊、张冰月、张景友、张旭、张志强等人也参与了本书的翻译工作，在此一并表示感谢！

由于水平有限，如有不妥之处，恳请读者指正。

前　言

本书可用作“数据结构与算法”课程的教材。“数据结构与算法”课程常常称为 CS2 课程，因为它通常作为计算机专业的第二门课程。

从教学的角度看，本书遵循了一流的 CS1 教材 *Java Software Solutions：Foundations of Program Design*（作者 John Lewis 与 William Loftus）一书的风格和方法，本书同样具备已经受到很高评价的一些特色，比如关键概念框和完整代码示例。作为计算机专业学生的两门或者三门导论课程，这两本书提供了一个坚实而系统的教学方案。也就是说，本书不要求学生在以前的课程中已经使用过 *Java Software Solutions：Foundations of Program Design* 一书。

在 CS1 和 CS2 课程中出现的内容（比如递归或排序），本书也同样进行了阐述。本书还给出了一些有用的参考材料，介绍了面向对象的概念以及如何在 Java 中实现。

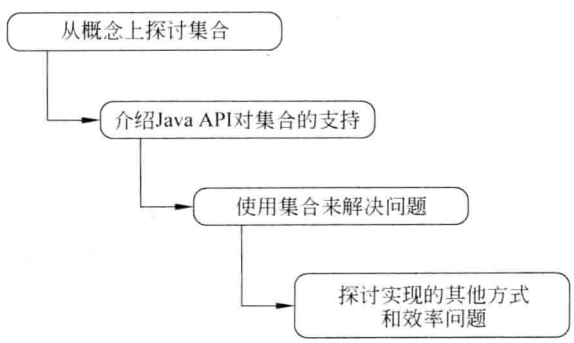
数据结构与算法在计算机专业课程中起着关键的作用，本书能较好地满足该课程的要求。

第 4 版的新内容

在第 4 版中，本书做了一些重要的修订，以得到更佳的教学效果：

- 修订了有关集合的章节，对 Java API 是如何支持集合的给出了更全面的阐述。
- 在每章的后面增加了术语和定义的汇总。
- 把迭代器单独成一章，并对它进行了扩展介绍。
- 增加了新的代码注解。
- 增加了新的常见错误图示。
- 增加了新的设计要点插图。
- 增加了一个新附录，介绍图形绘制、图形用户界面开发以及常规表达式。
- 对全书进行了修订和更新，以更详尽地介绍，并解决问题。

特别地，对每个集合的讨论进行了重新修改，以匹配如下流程：



这种方法阐明了 Java API 支持某个集合与从头开始实现集合之间的区别。它使得教师更容易指出 API 实现的局限性。这种方法还使得教师在需要的时候，只是简单地介绍一下集合，而不详细探讨其实现。

本版的其他改进使得本书在更高的层次上对内容进行介绍。每章后面增加的术语表（及其定义）提供了对核心问题的汇总。新的代码注释和常见错误特性，可以突出特定问题（否则，这些问题可能会淹没在正文中），且不会打断内容的连贯性。

我们认为，这些修订是建立在前面版本的教学基础之上的，为教师选择要教授的内容提供更好的灵活性。

本书的写作风格

这种类型的图书在整体写作方法上的差别相当大。本书的写作方法是建立在一些我们强烈推荐的重要原则之上的。首先，以一种连贯叙述的方式介绍在本书中将要考察的各种集合。其次，强调完美软件设计技巧的重要性。第三，对本书结构加以组织以支持和强化本书的重要目标：即数据结构与算法的学习。我们将更深入地考察这些原则。

本书始终坚持良好的软件工程实践准则，集合实现的设计以及使用这些设计的程序遵循一致且恰当的标准。

章 节 划 分

第 1 章（概述）讨论了软件质量涉及的各个方面，并对软件开发问题进行了一个全面概述。本章的目的在于，在进行数据结构和算法设计的细节之前，建立一套正确的开发思路。

第 2 章（算法分析）介绍了确定算法效率的基础知识，并阐述了一个重要的标准，使得开发人员可以以正确的方式将一种算法与另一种算法进行比较。本章的重点是理解重要的概念，而不是陷入数学或公式的漩涡。

第 3 章（集合概述——栈）建立了集合的概念，强调了将接口和实现区分开来的必要性。本章还从概念上介绍了栈，然后阐述了基于数组的栈的实现。

第 4 章（链式结构——栈）讨论了使用引用来创建链式数据结构。本章考察了有关链表管理的基本问题，然后使用基本链式数据结构（在第 3 章中已介绍），定义栈的另一种实现方法。

第 5 章（队列）考察了先进先出队列的概念和实现。本章通过一个高效使用队列的示例来讨论基数排序。本章所介绍的实现包括基本链表以及定长和环形数组。

第 6 章（列表）论述了 3 种类型的列表：有序表、无序表和索引表。通过讨论这 3 种类型列表所共有的及各自独有的操作，来对它们进行比较和区分。在各种类型的列表设计中，我们将恰当地使用继承，且通过两种方式（基于数组的和链式的表示）来实现这些列表。

第 7 章（迭代器）是新的一章，分别介绍了迭代器的概念和实现，这些对集合是非常

重要的。

第 8 章（递归）概要介绍了递归的概念，以及递归解决方案为什么是优美的。本章还探讨了递归的实现细节，并讨论了递归算法分析的基本思想。

第 9 章（排序与查找）讨论了线性查找和二分查找算法以及若干排序的算法（如选择排序、插入排序、冒泡排序、快速排序以及归并排序）。本章着重讨论与查找和排序相关的编程问题，比如将 Comparable 接口用作对象比较的基础，以特定数据结构为基础的查找与排序（如堆排序）将在后面的章节讨论。

第 10 章（树）对树进行了概述，并构建了关键的术语和概念。本章讨论了各种实现方式，并通过一个二叉树来表示和评估某一算数表达式。

第 11 章（二叉查找树）利用第 9 章构建的基本概念，定义了一个经典的二叉查找树。本章先考察了一个二叉查找树的链式实现，然后讨论了树结点的平衡如何对其性能起到关键作用。这就引出了 AVL 和二叉查找树的红/黑实现的介绍。

第 12 章（优先队列与堆）探讨了堆的概念、使用和实现，尤其是与优先队列的关系。我们通过一个堆排序来举例说明其用途。本章还介绍了链式实现和基于数组的实现。

第 13 章（Set 与 Map 集合）介绍了这两种类型的集合及其对 Java Collections API 的重要性。

第 14 章（多路查找树）是前面章节的自然延伸。本章讨论了 2-3 树、2-4 树以及广义 B 树的概念，还讨论了各种实现。

第 15 章（图）探讨了无向图和有向图的概念，并构建了一些重要的术语。本章考察了若干常用图的算法，并讨论了各种实现，包括邻接矩阵。

附录 A（UML）概述了统一建模语言。UML 是表示面向对象系统的事实标准表示法。

附录 B（面向对象设计）为任何需要回顾面向对象的基本概念及其在 Java 中如何实现等内容的读者提供参考。本附录包含的概念包括抽象、类、封装、继承、多态性以及许多相关的 Java 语言结构，比如接口。

附录 C（Java 图形编程）介绍了使用 Java API 绘制图形的基础知识。

附录 D（图形用户接口）详细介绍了开发基于 Swing 的 GUI 所需的元素。本附录还给出了使用各种接口组件的多种示例。

附录 E（散列）包括散列的概念及其相关问题，比如散列函数及冲突。本附录讨论了散列的各种 Java Collections API。

附录 F（正则表达式）介绍了正则表达式的使用，它在多种 Java API（如 Scanner 类）中发挥作用。

教 辅 资 料

本书的学生资源有：

- 本书所有程序的**源代码**。
- 本书某些内容的 VideoNotes。

这些资源可以从 http://www.pearsonhighered.com/lewis 获取。

在 Pearson Education 的教师资源中心有如下教辅资料：

- 本书中的部分练习题和编程项目的答案。
- 测试题库，含有测验用的问题。
- 讲解本书内容的 PowerPoint 幻灯片。

要获得这些资源，可以访问 http://www.pearsonhighered.com/irc，或与你当地的 Pearson Education 销售代表联系。

目　录

第 1 章　概述 1
1.1　软件质量 1
1.1.1　正确性 2
1.1.2　可靠性 2
1.1.3　健壮性 3
1.1.4　可用性 3
1.1.5　可维护性 3
1.1.6　可重用性 4
1.1.7　可移植性 4
1.1.8　运行效率 4
1.1.9　质量问题 5
1.2　数据结构 5
1.2.1　一个物理示例 5
1.2.2　以集装箱作为对象 7
关键概念 7
主要术语 8
自测题 8
练习题 8
自测题答案 9
第 2 章　算法分析 10
2.1　算法效率分析 10
2.2　增长函数与大 O 记法 11
2.3　增长函数的比较 13
2.4　时间复杂度分析 14
2.4.1　循环运行的复杂度分析 14
2.4.2　嵌套循环的复杂度分析 15
2.4.3　方法调用的复杂度分析 16
关键概念 17
主要术语 17
自测题 18
练习题 18
自测题答案 18
参考文献 19

第 3 章 集合概述——栈 20
3.1 集合 20
3.1.1 抽象数据类型 21
3.1.2 Java 集合 API 22
3.2 栈集合 23
3.3 主要的面向对象概念 24
3.3.1 继承与多态性 24
3.3.2 泛型 25
3.4 使用栈计算后缀表达式 26
3.4.1 Javadoc 32
3.5 异常 33
3.6 栈 ADT 34
3.7 用数组实现栈 36
3.7.1 管理容量 37
3.8 ArrayStack 类 38
3.8.1 构造函数 39
3.8.2 push 操作 40
3.8.3 pop 操作 41
3.8.4 peek 操作 42
3.8.5 其他操作 42
3.8.6 EmptyCollectionException 类 43
3.8.7 其他实现 43
关键概念 44
主要术语 44
自测题 45
练习题 45
程序设计项目 46
自测题答案 47
第 4 章 链式结构——栈 48
4.1 链接作为引用 48
4.2 管理链表 50
4.2.1 访问元素 50
4.2.2 插入结点 51
4.2.3 删除结点 51
4.3 无链接的元素 52
4.3.1 双向链表 52
4.4 Java API 中的栈 53
4.5 使用栈来穿越迷宫 53
4.6 用链表实现栈 61

4.6.1 LinkedStack 类 62
4.6.2 push 操作 65
4.6.3 pop 操作 65
4.6.4 其他操作 66
关键概念 67
主要术语 67
自测题 67
练习题 68
程序设计项目 68
自测题答案 68
第 5 章　队列 70
5.1 概述 70
5.2 Java API 中的队列 71
5.3 使用队列：代码密钥 72
5.4 使用队列：售票口模拟 75
5.5 队列 ADT 79
5.6 用链表实现队列 81
5.6.1 enqueue 操作 82
5.6.2 dequeue 操作 83
5.6.3 其他操作 84
5.7 用数组实现队列 84
5.7.1 enqueue 操作 87
5.7.2 dequeue 操作 89
5.7.3 其他操作 89
5.8 双端队列 89
关键概念 90
主要术语 90
自测题 90
练习题 90
程序设计项目 91
自测题答案 92
第 6 章　列表 93
6.1 列表集合 93
6.2 Java 集合 API 中的列表 95
6.3 使用无序列表：学习计划 95
6.4 索引列表使用示例：Josephus 问题 105
6.5 列表 ADT 107
6.5.1 往列表中添加元素 107
6.6 使用数组实现列表 112

6.6.1 remove 操作 113
6.6.2 contains 操作 115
6.6.3 有序列表的 add 操作 116
6.6.4 无序列表的特有操作 117
6.6.5 无序列表的 addAfter 操作 117
6.7 使用链表实现列表 118
6.7.1 remove 操作 119
关键概念 120
主要术语 120
自测题 121
练习题 121
程序设计项目 122
自测题答案 123
第 7 章 迭代器 124
7.1 何谓迭代器 124
7.1.1 迭代器的其他问题 126
7.2 使用迭代：再论学习计划程序 126
7.2.1 显示某些课程 130
7.2.2 删除课程 131
7.3 用数组实现迭代器 132
7.4 用链表实现迭代器 135
关键概念 136
主要术语 137
自测题 137
练习题 137
自测题答案 137
第 8 章 递归 138
8.1 递归地思考 138
8.1.1 无穷递归 139
8.1.2 数学中的递归 140
8.2 递归地编程 140
8.2.1 递归与迭代 142
8.2.2 直接递归与间接递归 143
8.3 使用递归 143
8.3.1 穿越迷宫 143
8.3.2 汉诺塔 150
8.4 递归算法分析 154
关键概念 155
主要术语 156

自测题 156
练习题 156
程序设计项目 157
自测题答案 158
第 9 章　排序与查找 159
9.1　查找 159
9.1.1　静态方法 160
9.1.2　泛型方法 160
9.1.3　线性查找法 161
9.1.4　二分查找法 162
9.1.5　查找算法的比较 164
9.2　排序 165
9.2.1　选择排序法 168
9.2.2　插入排序法 170
9.2.3　冒泡排序法 171
9.2.4　快速排序法 173
9.2.5　归并排序法 175
9.3　基数排序法 178
关键概念 181
主要术语 182
自测题 183
练习题 183
程序设计项目 184
自测题答案 185
第 10 章　树 186
10.1　概述 186
10.1.1　树的分类 187
10.2　实现树的策略 188
10.2.1　树的数组实现之计算策略 188
10.2.2　树的数组实现之模拟链接策略 189
10.2.3　树的分析 190
10.3　树的遍历 190
10.3.1　前序遍历 191
10.3.2　中序遍历 191
10.3.3　后序遍历 192
10.3.4　层序遍历 193
10.4　二叉树 193
10.5　使用二叉树：表达式树 197
10.6　背部疼痛诊断器 208

10.7 用链表实现二叉树 212
10.7.1 find 方法 216
10.7.2 iteratorInOrder 方法 217
关键概念 218
主要术语 219
自测题 220
练习题 220
程序设计项目 220
自测题答案 221
第 11 章 二叉查找树 222
11.1 概述 222
11.2 用链表实现二叉查找树 224
11.2.1 addElement 操作 225
11.2.2 removeElement 操作 227
11.2.3 removeAllOccurrences 操作 231
11.2.4 removeMin 操作 232
11.3 用有序列表实现二叉查找树 233
11.3.1 BinarySearchTreeList 实现的分析 236
11.4 平衡二叉查找树 237
11.4.1 右旋 238
11.4.2 左旋 238
11.4.3 右左旋 238
11.4.4 左右旋 239
11.5 实现二叉查找树：AVL 树 239
11.5.1 AVL 树的右旋 240
11.5.2 AVL 树的左旋 240
11.5.3 AVL 树的右左旋 241
11.5.4 AVL 树的左右旋 241
11.6 实现二叉查找树：红黑树 242
11.6.1 红黑树中的元素插入 242
11.6.2 红黑树中的元素删除 244
关键概念 246
主要术语 246
自测题 247
练习题 247
程序设计项目 248
自测题答案 248
参考文献 250

第 12 章　优先队列与堆.....251
12.1　堆.....251
12.1.1　addElement 操作.....253
12.1.2　removeMin 操作.....254
12.1.3　findMin 操作.....255
12.2　使用堆：优先级队列.....255
12.3　用链表实现堆.....259
12.3.1　addElement 操作.....260
12.3.2　removeMin 操作.....262
12.3.3　findMin 操作.....265
12.4　用数组实现堆.....265
12.4.1　addElement 操作.....267
12.4.2　removeMin 操作.....268
12.4.3　findMin 操作.....270
12.5　使用堆：堆排序.....270
关键概念.....271
主要术语.....272
自测题.....272
练习题.....273
程序设计项目.....273
自测题答案.....274
第 13 章　Set 与 Map 集合.....275
13.1　概述.....275
13.2　Java API 中的 Set 和 Map.....276
13.3　使用 Set 集合：域阻塞器.....278
13.4　使用 Map 集合：产品销售.....281
13.5　使用 Map 集合：用户管理.....284
13.6　使用树来实现 Set 和 Map 集合.....288
13.7　使用散列来实现 Set 和 Map 集合.....289
关键概念.....291
主要术语.....292
自测题.....292
练习题.....292
程序设计项目.....293
自测题答案.....293
第 14 章　多路查找树.....294
14.1　整合树的概念.....294
14.2　2-3 树.....294
14.2.1　往 2-3 树中插入元素.....295

14.2.2 从 2-3 树中删除元素 297
14.3 2-4 树 299
14.4 B 树 300
14.4.1 B*树 301
14.4.2 B+树 301
14.4.3 B 树的分析 302
14.5 B 树的实现策略 302
关键概念 303
主要术语 303
自测题 304
练习题 304
程序设计项目 304
自测题答案 305
参考文献 305
第 15 章 图 306
15.1 无向图 306
15.2 有向图 308
15.3 网络 309
15.4 常用的图算法 310
15.4.1 遍历 310
15.4.2 测试连通性 313
15.4.3 最小生成树 314
15.4.4 判定最短路径 317
15.5 图的实现策略 317
15.5.1 邻接列表 317
15.5.2 邻接矩阵 318
15.6 用邻接矩阵实现无向图 319
15.6.1 addEdge 方法 323
15.6.2 addVertex 方法 323
15.6.3 expandCapacity 方法 324
15.6.4 其他方法 325
关键概念 325
主要术语 325
自测题 326
练习题 326
程序设计项目 327
自测题答案 327
参考文献 328

附录 A　UML 329
A.1　统一建模语言 329
A.2　UML 类图 329
A.3　UML 关系 330
关键概念 332
自测题 333
练习题 333
自测题答案 333
附录 B　面向对象设计 334
B.1　概述 334
B.2　使用对象 335
B.2.1　抽象 335
B.2.2　创建对象 336
B.3　类库与包 337
B.3.1　import 声明 338
B.4　状态与行为 338
B.5　类 339
B.5.1　实例数据 342
B.6　封装 342
B.6.1　可见性修饰符 343
B.6.2　局部数据 344
B.7　构造函数 344
B.8　方法重载 345
B.9　再谈引用 346
B.9.1　空引用 346
B.9.2　this 引用 347
B.9.3　别名 348
B.9.4　垃圾回收 350
B.9.5　将对象作为参数传递 350
B.10　static 修饰符 351
B.10.1　静态变量 351
B.10.2　静态方法 352
B.11　包装类 352
B.12　接口 353
B.12.1　Comparable 接口 355
B.12.2　Iterator 接口 355
B.13　继承 355
B.13.1　派生类 356
B.13.2　protected 修饰符 358

B.13.3 super 引用 358
B.13.4 重载方法 359
B.14 类的层次结构 359
B.14.1 Object 类 360
B.14.2 抽象类 361
B.14.3 接口的层次结构 362
B.15 多态性 363
B.15.1 引用和类的层次结构 363
B.15.2 基于继承的多态性 364
B.15.3 基于接口的多态性 365
B.16 异常 367
B.16.1 异常消息 367
B.16.2 try 语句 368
B.16.3 异常传播 369
B.16.4 异常类的层次结构 369
关键概念 370
自测题 372
练习题 373
程序设计项目 373
自测题答案 375
附录 C Java 图形编程 376
C.1 像素与坐标 376
C.2 颜色表示 376
C.3 绘制图形 377
C.4 多边形与多直线 384
C.4.1 Ploygon 类 387
练习题 388
程序设计项目 388
附录 D 图形用户接口 391
D.1 GUI 的构成 391
D.1.1 框架与面板 394
D.1.2 按钮与动作事件 395
D.1.3 确定事件源 396
D.2 更多 GUI 组件 399
D.2.1 文本域 399
D.2.2 核选框 402
D.2.3 单选按钮 405
D.2.4 滑块 408
D.2.5 组合框 412

D.2.6　计时器 416
D.3　布局管理器 420
D.3.1　流水布局管理器 422
D.3.2　边界布局管理器 424
D.3.3　网格布局管理器 426
D.3.4　方框布局管理器 428
D.3.5　包容分层结构 430
D.4　鼠标与按键事件 430
D.4.1　鼠标事件 430
D.4.2　按键事件 437
D.4.3　扩展适配器类 441
D.5　对话框 441
D.5.1　文件选择器 443
D.5.2　颜色选择器 446
D.6　一些重要细节 446
D.6.1　边框 446
D.6.2　工具提示与助记符 450
D.7　GUI 设计 455
关键概念 455
自测题 456
练习题 457
程序设计项目 457
自测题答案 459
附录 E　散列 461
E.1　概述 461
E.2　散列函数 463
E.2.1　余数法 463
E.2.2　折叠法 463
E.2.3　平方取中法 464
E.2.4　基数转换法 464
E.2.5　数字分析法 464
E.2.6　长度相关法 465
E.2.7　Java 语言中的散列函数 465
E.3　解决冲突 465
E.3.1　链地址法 466
E.3.2　开放地址法 467
E.4　从散列表删除元素 469
E.4.1　从链地址实现中删除 470
E.4.2　从开放地址实现中删除 470

E.5 Java 集合 API 中的散列表 471
E.5.1 Hashtable 类 471
E.5.2 HashSet 类 472
E.5.3 HashMap 类 473
E.5.4 IdentityHashMap 类 474
E.5.5 WeakHashMap 类 475
E.5.6 LinkedHashSet 与 LinkedHashMap 476
关键概念 477
自测题 477
练习题 478
程序设计项目 478
自测题答案 479
附录 F 正则表达式 481

第1章 概 述

本章学习目标

- 明确软件质量的各个方面
- 基于软件质量问题，引导出数据结构
- 介绍数据结构的基本概念
- 介绍几种基本的数据结构

本章首先围绕软件系统的基本问题进行概述，然后再介绍数据结构的内容。软件系统按需工作很有必要，但只是简单的工作还不够，理解这一点很重要。我们还必须开发高质量的软件系统。本章将介绍与软件质量和数据结构有关的各种问题，并给出一些术语，这些术语对数据结构和软件设计的探讨是至关重要的。

1.1 软 件 质 量

设想这样一个场景：你正走向大河上最近修建的一座桥。当你靠近这座桥时，你会看到一个标牌，它告诉你，这座桥是由当地建筑工人设计并修建的，该工程没有工程师的参与。你会打算继续过这座桥吗？如果该标牌告诉你，这座桥是由工程师设计，然后由建筑工人建造的话，那么情况会有所不同吗？

这里的“工程师”一词指的是受过工程学历史、理论、方法与实践教育的个人。该定义包括诸如电子工程、机械工程及化学工程等研究领域。**软件工程**（Software Engineering）是一门关于高质量软件开发的技术和理论的学科。

“软件工程”这一术语是在20世纪70年代，该行业的领先者认识到本行业创建的许多软件的质量都是甚为低劣的时候提出的。他们希望开发人员不是简单地编写程序，而是转向软件工程这种更为专业的思想。要构建软件，我们首先必须明白，该术语不仅仅是一个名称，事实上它代表的是一种完全不同的态度。

对于软件工程是否已经成为一门真正的工程科学还存在许多争论。我们把这些争论留到软件工程课程中去讨论。这里，我们需要明白的是，作为软件开发人员，我们具有相同的历史，受相同的理论约束，必须理解当前的方法和实践以便能一同工作。

最后，我们希望能让客户满意，这些客户包括为所开发软件付费的个人或组织，以及该系统的最终用户。

软件工程的目标很大程度上与其他工程学科是一样的。

- 解决正确性问题。
- 按时且在预算之内给出解决方案。
- 给出高质量的解决方案。

- 以合情合理的方式完成上面事情（请参见 www.acm.org/about/code-of-ethics）。

牛顿说过，“我之所以看得更远，是因为我站在巨人的肩膀上。”作为现代的软件开发人员，我们是站在创建和开发软件领域的巨人的肩膀上。要真正站在他们的肩膀上，我们必须理解合格软件系统的基础知识以及数据的组织、存储和检索。这些是创建现代软件开发的构建模块。

为了最大限度地提高软件质量，我们首先必须认识到，软件质量对于不同客户有不同的含义。要考虑的软件质量特征有很多。图 1.1 列出了高质量软件的几个特征。

质量特征	描　述
正确性	软件在多大程度上满足其特定需求
可靠性	软件故障发生的频率和危险程度
健壮性	出错情况下可以得到恰当处理的程度
可用性	用户学习和执行任务的难易程度
可维护性	对软件进行修改的难易程度
可重用性	软件组件可重用于其他软件系统开发的难易程度
可移植性	软件组件可以在多个计算机环境下使用的难易程度
运行效率	在不浪费资源的情况下软件完成其目标的程度

图 1.1　软件质量的特征

1.1.1　正确性

正确性（Correction）的概念源于我们开发出恰当的解决方案这一初始目标。在软件开发道路的每一步上，我们都要确保，我们是按照需求规范所规定的那样在处理问题，并为该问题提供了一个精确的解决方案。如果软件没有解决正确性问题，那么软件质量的其他所有方面几乎都是毫无意义的。

1.1.2　可靠性

你是否曾经试图以电子形式访问你的银行账号而无法成功，你是否曾经因为你所使用的软件或硬件的故障而损失了全部的工作成果，那么你对**可靠性**（Reliability）的概念就已经有深刻体会了。软件**故障**（failure）可以这样定义：在可允许的操作环境中发生的任何无法容忍的行为。我们可以计算出可靠性的度量值，比如故障发生的平均时间，或发生两个故障之间的操作或任务数量。

关键概念

可靠的软件很少发生故障，即使发生了故障，也可以将该故障的影响降到最低。

在一些情形下，可靠性是生死攸关的问题。在 20 世纪 80 年代早期，人们设计出一种被称为 Therac-25 的医疗设备，它可以按照技师通过一种特殊键盘设定的值来进行放射治

疗。但控制该设备的软件存在一个错误，当技师在键盘上做出一个非常精细的调整后，该设备的内部设定却被彻底改变并发射出致命的辐射量。由于该故障的发生频率没有规律，以至于在确定问题的根源之前已经造成了若干人的死亡。

在其他一些情况下，可靠性问题还能带来财务方面的损失。在 1998 年 11 月的某天，美国东部整个 AT&T 公司网络设施发生了故障，导致了严重的通信中断。该问题最终归结于某个特殊的软件错误。该故障给受影响的公司造成了几百万美元的损失。

1.1.3　健壮性

健壮性（Robustness）是用来衡量一个系统有多么健壮的。一个健壮的系统可以完美地处理各种问题。比如，某个特定的输入域是用来处理数字数据的，如果键入了字母信息会怎样呢？该程序会因为错误的发生而异常终止。然而，一个更健壮的解决方案应该设计成这样：系统能够识别和处理该情形，并给出恰当的错误消息。

开发一个完全健壮的系统也许能值得上开发成本，也许不能。在某一情况下，如果发生很不正常的情形，程序会异常终止，这是完全可以接受的。另一方面，如果增加像此类的保护不会带来过分开销，这绝对是很好的开发习惯。而且，良好定义的系统需求应该详细且清楚地描述在哪些环境下要求有健壮的错误处理。

1.1.4　可用性

软件系统要想是有效率的，就必须是确实可用的。如果某个系统很难使用，那么它提供的功能再好也是毫无意义的。在计算机科学的学科中，有一个称为**人机交互**（Human-Computer Interaction，HCI）的研究领域，该研究领域主要研究软件系统的用户界面的分析与设计。用户与系统间的交互必须设计优良，包括帮助选项、能给出有意义的消息、界面布局一致、颜色的使用恰当、错误预防以及错误恢复。

1.1.5　可维护性

软件开发人员必须对他们的软件进行**维护**。也就是说，为了修复错误、提升系统的功能或跟上发展的需要，他们必须对软件进行修改。一个实用的软件系统，在最初开发完成之后，可能需要多年的维护。进行维护任务的软件工程师，往往不是开发该软件的工程师。因此，为了使软件系统的可维护性最大化，软件系统必须具有良好的结构、良好的编码和良好的文档说明是非常重要的。

关键概念

软件系统必须经过细心设计、编码和文档说明，以便为开发人员、维护人员和用户的工作提供支持。

大型的软件系统很少是由单个人甚至是一个开发小组来编写的。相反，系统的开发通常都是由广泛分布于不同地点的大型团队来共同完成的。基于此，开发人员之间的通信交

流尤为关键。因此，创建可维护性的软件，不仅是出于长期考虑，对最初的开发工作也是有益的。

1.1.6 可重用性

假设你是建造一栋办公楼的一名承包商，你可能会从零开始设计并制造该建筑内的每一扇门。这将需要大量的设计和建造工作，毫无疑问还需要大量的金钱。另外一个选择是，使用事先设计并预制好的门安装到该建筑的门框上。这种方式意味着你可以节省大量时间和金钱，因为你可以使用以前已经使用过很多次且被证明是可靠的设计。你可以确信该设计已经经过了充分检验，并且你也了解它的性能。然而，这并不排除这种一种可能性，即建筑内有些门需要专门的设计和建造以满足某一特殊要求。

在开发软件系统的过程中，如果已有的软件组件能满足开发人员和客户的要求，则使用这些软件组件往往是有意义的。为什么要去重复发明轮子呢？已有的软件组件可以是从整个子系统到单个的类和方法。它们可能来自早期开发的另一系统，也可能来自人们创建的用于支持未来系统开发的组件库。有些已有的组件称为**商用现货**（Commercial Off-The-Shelf，COTS）产品。因为已有组件常常都已经在其他系统中检测过了，所以它们通常都是可靠的。

使用已有组件能够减少软件开发工作量。然而，重用是要付出代价的。开发人员必须花时间对已有的组件进行调查以找出合适的组件。该组件通常需要经过修改或扩展以满足新系统的要求标准。因此，如果该组件是真正可重用的，那么它是很有帮助的。因此，软件必须进行良好编码和文档说明，以便能容易整合到新系统中，且易于修改或扩展以满足新的需求。

1.1.7 可移植性

可移植性好的软件，可以毫不费力或费力很少地从一种计算环境移植到另一种计算环境。使用某一特定操作系统和中央处理器（CPU）开发的软件，在另一种环境下可能运行不好或完全不能运行。一个显而易见的问题是，程序是被编译成了某一特定 CPU 机器语言。因为每种类型的 CPU 都有自己的机器语言，将其移植到另外一台机器时需要另外一种已翻译的版本。各种翻译之间的差别可能会导致“同一”程序在两种不同类型的机器上运行的不同。

使用 Java 语言就可以解决这个问题，因为 Java 的源代码被编译成了**字节码**（bytecode），这是一种低级语言，它不是任何特定 CPU 的机器语言。字节码运行在 Java **虚拟机**（Java Virtual Machine，JVM）上。JVM 是一种解释并执行字节码的软件。因此，至少从理论上讲，任何具有 JVM 的系统都能运行 Java 程序。

1.1.8 运行效率

关键概念

软件必须高效地使用诸如 CPU 时间和存储器之类的资源。

图 1.1 中所示的质量特征最后一项是运行效率。软件系统应该有效地利用分配给它们的资源。两个主要的资源是 CPU 时间和内存。用户对计算机及其软件的要求，从计算机被创造出的第一天起就在不断提高。软件必须最大化利用资源以满足这些要求。单个算法的运行效率就是这个问题的一个重要部分，在下一章乃至全书将会更详细地讨论这个问题。

1.1.9　质量问题

关键概念

必须优先考虑质量特征，并尽最大努力实现。

在一定程度上讲，软件质量体现在旁观者的眼中。也就是说，对一些人来说，软件质量更为重要。我们必须考虑各种所有人的需要，这些人以这样或那样的形式受软件项目的影响。例如，最终用户当然希望最大化软件的可靠性、可用性和运行效率，但未必在意其可维护性或可重用性。客户不仅要确保满足用户，而且还关心软件的总成本。开发人员和维护人员则希望内部系统具有较高的质量。

同样要注意的是，某些质量特征彼此之间有时是互斥的。例如，为了提高某个程序的运行效率，可能要选用一个较难理解的复杂算法，因而难以维护。这就要求我们必须认真地把某一特定项目的相关问题排列出一个优先次序，然后在这些范围内，尽可能地最大限度地满足所有质量特征。如果为了运行效率而必须使用更为复杂的算法，那么，为了便于日后的维护工作，应特别地对这些代码加上文档说明。

1.2　数 据 结 构

为什么要在一本有关数据结构及其算法的教材中，花费这么多的时间来讨论软件工程和软件质量呢？原因是，当你开发一个复杂的程序时，设计一个更为成熟的开发过程规划是很重要的。正如本章开始介绍的那样，软件开发的目的是构建软件，而不仅仅是编写代码。本书中要介绍的数据结构，将为复杂软件开发打下基础。下面通过一个示例来说明为什么需要数据结构以及各种数据结构方法。

1.2.1　一个物理示例

假设你需要设计这样一个处理过程：如何把集装箱从货船上卸载到码头。在比较理想的情况下，当货船要卸货时，用于把集装箱拉到目的地的卡车正好在那里等待着，这样就无需储存集装箱了。但是，往往不可能是这么准时的，也无需卡车有这么高的效率，因为当每条货船卸货时，它们不得不站在那里等待。实际情况是，每个集装箱从货船上卸下来后，运到一个储存地储存起来，直到装上卡车拉到目的地。我们的目标是，使得集装箱的卸载和集装箱等待运走的时间尽可能短。这就意味着在每个过程中所需要的查找工作量最小，以及集装箱移动的次数最少。

在进一步深入这个问题之前，先来看看这个问题的初始假设。首先，集装箱是相同的。这意味着它们有着相同的形状、相同的大小，可以容纳相同的体积和类型的货物。第二，每个集装箱具有唯一的标识码。该标识码决定了每个集装箱的最终目的地，以及是否用卡车来运输。第三，码头上负责处理集装箱的工作人员无需知道、也不能知道每个集装箱中装的是什么东西。

以这些假设为出发点，我们应如何为这些集装箱设计存储处理？一种简单的方法是，设置一个很大的数组，该数组以集装箱的标识码为索引。但是，要记住，不仅是对某一条货船，对所有货船和所有集装箱，标识码都是唯一的。这意味着需要在每个码头设置一个足够大的存储数组，以存储来自全世界的集装箱。这同时也意味着，在某个时间里，在这些物理数组中的绝大部分单元是空的。显然，这是可怕的空间浪费，效率也非常低。下一章将探讨算法效率的问题。

仍可使用与尝试上面相同的解决方法，只不过这里允许当有集装箱运走或运来时，数据可以像 ArrayList 一样缩小或扩大。但是，给定这个数组，当有标识码更小的集装箱运来或运走时，这种解决方法需要移动集装箱多次。同样，这种解决方法的效率也不高。

如果能估计出某个时候在码头上将要存储的集装箱的最大数目，情况会是怎样呢？这就可以创建一个具有这个最大数目的数组，然后就可以把从货船上卸载下的集装箱放置到数组中下一个可用的位置上。对卸载来说，这是相对有效的，因为码头的工作人员只需跟踪数组中为空的位置，并把新运来的集装箱放置到空位置上即可。但是，使用这种方法时，如果卡车司机要查找某个集装箱怎么办？由于数组不是按标识码排序的，卡车司机就得在数组中查找那个集装箱。在最坏的情况下，卡车司机要查找的集装箱是数组的最后一个，因此他们必须搜索整个数组。当然，平均情况是，每个卡车司机需要搜索半个数组。同样，看上去这也不够有效。

在尝试一种新的设计之前，让我们来重新看看每个集装箱。当然，有了集装箱的标识码。从标识码中，我们也知道了每个集装箱的目的地。如果不是用一维数组对集装箱排序，而是二维数组，情况会如何？这样我们就可以这样来实现：数组的第一维表示目的地。卸载集装箱仍很简单，码头工作人员只要按目的地把集装箱放置到数组的第一个空位置中即可。然后，卡车司机就无需查找整个码头了，而是只需查找表示目的地的数组部分。这种方法类似于一种称为**散列表**（hash table）的数据结构（参见附录 E），其效率有所提高，但仍需要至少查找存储中的一部分。

这种解决方法的第一部分（即按目的地来组织数组的第一维）看上去是一个好主意。但是，对第二维使用一个无序的数组仍无法满足要求。对每个集装箱，还有另外一个需要考察的信息，那就是集装箱从货船上卸载下来的顺序。考虑这样一个示例，一个石油钻塔的各个零部件装在多个集装箱中。这些集装箱的顺序重要吗？是的。组装钻塔的工作人员在组装钻探的顶部之前，必须先拿到并组装钻塔的基部。此时，集装箱从货船运走的顺序以及放置到存储点的顺序是重要的。这里只考虑了集装箱的卸载、存储和运输问题。现在我们在更大的环境下来看看这个问题，包括货船在原始点是如何装载的。货船的装载有三种可能性：集装箱按钻塔部件的组装顺序装载，集装箱按反序装载，或者集装箱没有按顺序但按集装箱的标识码相关信息进行装载。下面来分别看看这些情况。

记住，卸载货船的过程与装载货船的顺序正好相反。这很像数据结构**栈**（stack）的操

作，第 3 章和第 4 章将进一步介绍它。如果是按钻塔部件的组装顺序装载货船，那么卸载过程则正好相反。于是，存储和检索过程必须颠倒顺序，以便使集装箱恢复到正确的顺序。这可以这样来完成：按目的地把集装箱放置到数组中，并按卸载顺序存储。然后，卡车司机就可以从最后一个开始检索这些集装箱。最终，利用这种解决方法，我们的存储和检索过程就根本不需要查找和其他另外的操作。

如果是按钻塔部件的组装相反的顺序来装载货船，情况会如何呢？在这种情况下，卸载货船会把集装箱变回到所需要的顺序，存储和检索过程必须保持这种顺序。这可以这样来完成：按目的地把集装箱放置到数组中，并按卸载顺序存储。卡车司机就可以从第一个而不是最后一个开始检索这些集装箱。同样，利用这种解决方法，存储和检索过程也不需要查找和其他另外的操作。这就是数据结构**队列**（queue）的操作，在第 5 章将详细介绍。

关键概念

队列可以保持其数据的顺序。

上面两种解决方法的效率很高，但与集装箱装载到货船上的顺序有关系。如果集装箱不是以某种特定顺序装载到货船的，情况会是如何呢？在这种情况下，需要按集装箱的优先级将目的地数组进行排序。有多种数据结构可以用来实现这个目的，包括有序列表（第 6 章）、优先队列与堆（第 12 章）以及 Map（第 13 章）。这里还有一个问题，那就是最好以何种方式组织一个文件，把集装箱的标识码与每个集装箱的其他信息关联？这个问题的解决方法可能包含了二叉查找树（第 11 章）和多路查找树（第 14 章）。

1.2.2　以集装箱作为对象

货运集装箱示例阐述了数据结构的另一个问题，那就是如何存储这些集装箱。它同时还阐述了集装箱存储什么的问题。在前面的讨论中，假设所有货运集装箱具有相同的形状和相同的大小，可以容纳相同体积和类型的货物。为了能进行相互交换的存储和运输，前两个假设必须保持为真，但后面的则不必。

例如，考虑要运输的货物冷藏的。为此就要开发冷藏的运输集装箱。同样，一些货物要求有非常严格的湿度控制，因而需要集装箱还要有环境控制功能。另外一下集装箱还应设计为可以容纳危险品，并需要一层或双层的外围防护。一旦设计了这么多种集装箱，那么集装箱的种类就不再是相同的了。我们就可以把货运集装箱看成一种分层结构，就像面向对象程序设计中的类结构。这样，把货物装入货运集装箱，就类似于把对象赋给一个引用。货运集装箱和对象一样，并不是所有的赋值都是合法的。我们将在第 3 章中讨论兼容性、继承、多态性以及创建泛型数据结构的问题。

关键概念

- 可靠的软件很少发生故障，即使发生了故障，也可以将该故障的影响降到最低。
- 软件系统必须经过精心设计、编码和文档说明，以便为开发人员、维护人员和用户

提供支持。

- 软件必须有效使用诸如 CPU 时间和内存之类的资源。
- 质量特征必须优先考虑，并尽可能地最大化。
- 栈可用于颠倒数据集的顺序。
- 队列可以保持其数据的顺序。

主要术语

bytecode：字节码 软件的底层表示，运行在 Java 虚拟机上。

correctness：正确性 一种软件质量特性，表明软件达到特定需求的程度。

efficiency：效率 一种软件质量特性，表明软件实现其目标而无浪费资源的程度。

maintainability：可维护性 一种软件质量特性，表明对软件进行修改的容易程度。

portability：可移植性 一种软件质量特性，表明软件组件可用于多种计算机环境中的容易程度。

reliability：可靠性 一种软件质量特性，表明软件发生故障的频率和危害程度。

reusability：可重用性 一种软件质量特性，表明软件组件可被其他软件系统开发使用的容易程度。

robustness：健壮性 一种软件质量特性，表明能很好处理错误情况的程度。

software failure：软件故障 软件系统在可允许的操作条件下，发生的不可预知行为。

stakeholder：股东 投资一个项目的人，因此他关注与项目相关的特定问题。

usability：可用性 一种软件质量特性，表明用户在软件中掌握和运行任务的容易程度。

自测题

SR 1.1 软件工程与程序设计的区别是什么？

SR 1.2 请说出几个软件质量特征。

SR 1.3 可靠性与健壮性之间的关系是什么？

SR 1.4 请描述一下开发维护性好的软件的益处。

SR 1.5 Java 程序设计语言是如何解决可移植性的质量特征的？

SR 1.6 栈与队列操作之间的基本不同是什么？

SR 1.7 请说出两个用于给数据集排序的常用数据结构。

练习题

EX 1.1 比较并对比软件工程与其他工程学科。

EX 1.2 举出一个具体的例子，说明图 1.1 中列出的全部软件质量特征。

EX 1.3 给出一个示例，描述一下在软件系统开发中，可能需要一些质量特征进行折中。

自测题答案

SRA 1.1　软件工程关注的是系统设计和开发这一更大的目标，而不仅仅是编写代码。随着程序员们开始理解与高质量软件开发相关的问题，并开始采用合适的实践方法，他们也就成为了一名成熟的软件工程师。

SRA 1.2　软件质量特征包括：正确性、可靠性、健壮性、可用性、可维护性、可重用性、可移植性以及运行效率。

SRA 1.3　可靠性关注的是软件系统发生故障的频率以及在什么环境下发生故障，而健壮性关注的是软件系统出现故障时会发生什么。

SRA 1.4　结构良好、设计良好以及文档说明良好的软件更容易维护。这些特征对分布式开发也很有好处。

SRA 1.5　Java 程序设计语言是这样来解决这个问题的：把 Java 的源代码编译成了字节码，这是一种低级语言，它不是任何特定 CPU 的机器语言。字节码运行在 Java 虚拟机上。Java 虚拟机是一种解释并执行字节码的软件。因此，至少从理论上讲，任何具有 Java 虚拟机的系统都能运行 Java 程序。

SRA 1.6　栈会颠倒数据集的顺序，而队列则可以保持数据集的顺序。

SRA 1.7　可用于给数据集排序的常用数据结构有有序列表、堆和散列表。

第2章 算法分析

本章学习目标

- 讨论高效率软件的开发
- 介绍算法分析的概念
- 探讨渐进复杂度的概念
- 比较各种增长函数

在创建数据结构之前，理解算法效率的概念非常重要。正确创建并能高效利用 CPU 和内存的数据结构，可以在很多不同的应用程序中有效地重复使用。如果创建的数据结构效率不高，那么就像使用一份损坏的原作品来复印一样。

2.1 算法效率分析

软件质量的特征之一是资源的使用效率。CPU 的使用时间是最重要的资源之一。为完成某一特定任务所使用的算法的效率，是决定一个程序运行速度的主要因素。尽管用于算法分析的技术与所使用的内存大小有关，但这里讨论的重点是处理时间的使用效率。

算法分析（analysis of algorithms）是计算机科学的一个基础，并涉及多种技术和概念。它是贯穿此书的一个首要主题。本章介绍了与算法分析有关的一些问题，并为算法分析技巧的使用奠定了基础。

关键概念

算法分析是计算机科学的基础。

让我们以一个日常例子开始：手洗盘子。假设洗一个盘子要花费 30 秒，而烘干一个盘子又要再花费 30 秒，那么，我们很容易知道，洗完并烘干 n 个盘子需花费 n 分钟。该计算过程可以表示如下：

Time (n 个盘子)＝n×(30 秒洗盘时间＋30 秒烘干时间)＝60n 秒

或更正式地写作：

$$f(x)=30x+30x$$

$$f(x)=60x$$

另外，假设我们在洗盘子的时候不够小心，在周围溅了太多水。假设每洗一个盘子时，不仅要烘干这个盘子，而且还要烘干在此之前洗过的所有盘子。每洗一个盘子仍需花费 30 秒，但是现在将需花费 30 秒来烘干最后洗的那个盘子（一倍），花费 2×30 即 60 秒来烘干倒数第二个洗的盘子（两倍），花费 3×30 即 90 秒的时间来烘干倒数第三洗的盘子（三倍），

依此类推。计算过程可以表示如下：

$$\text{Time (n 个盘子)} = n \times (30 \text{ 秒洗盘时间}) + \sum_{i=1}^{n} (i \times 30)$$

使用算术等式 $\sum_{i=1}^{n} i = n(n+1)/2$，那么上面的函数变为：

$$\text{Time (n 个盘子)} = 30n + 30n\,(n+1)/2 = 15n^2 + 45n \text{ 秒}$$

如果有 30 个盘子要洗，第 1 种方法花费时间为 30 分钟，但是第 2 种（粗心）方法却需花费 247.5 分钟。洗的盘子越多，这种差距越明显。

2.2　增长函数与大 O 记法

对于每一个要分析的算法，都需要定义该问题的大小。就洗盘子的例子而言，该问题的大小是需要洗完并烘干的盘子数。还必须确定一个表示有效使用时间或空间的值。从时间上考虑，通常挑选一个恰当的最少处理步骤，比如，使洗和烘干盘子的次数最小。该任务所花费的总时间量，与执行该任务的次数直接相关。算法的效率可以根据该问题的大小和处理步骤数来定义。

考虑把一串数按升序进行排序的算法，一种表达该问题大小的自然方式是需要排序的值的数量。最优化的处理步骤可以表示为该排序算法做排序时需要的比较次数。做的比较次数越多，所使用的 CPU 时间也越多。

关键概念

增长函数表示与该问题大小相对应的时间或空间的使用。

增长函数（growth function）表示问题（n）大小与我们希望最优化的值之间的关系。该函数表示了该算法的**时间复杂度**（time complexity）或**空间复杂度**（space complexity）。

第 2 种洗盘子算法的增长函数是：

$$t(n) = 15n^2 + 45n$$

但是，并不总是需要知道某一算法确切的增长函数。相反，我们主要感兴趣的是算法的**渐进复杂度**（asymptotic complexity）。这就是说，我们主要关注随着 n 的增加时增长函数的一般性质。这一特性基于该表达式的主项，即 n 增加时表达式中增长最快的那一项。随着 n 变得非常大，洗盘子增长函数的值趋近于 n^2，因为 n^2 项的增长远快于 n 项。随着 n 的增长，常量（这里是 15 和 45）和次要项（这里是 45n）很快就可以忽略不计了。

图 2.1 中的表显示了两个项与表达式值是如何增长的。从该表可以看到，当 n 变大时，$15n^2$ 项决定着该表达式的值。注意，对于很小的 n 值，$15n^2$ 项比 45n 项更大。这就是说，当 n 变大时，某个主项并不一定对 n 的所有值都是比其他项更大。

渐进复杂度称为算法的**阶次**（order）。这样，第二个洗盘子算法具有阶次为 n^2 的时间复杂度，记为 $O(n^2)$。第一个洗盘子的示例，其增长函数为 $t(n)=60n$，其阶次为 n，记为 O(n)。因此，O(n)算法与 $O(n^2)$算法的不同在于，每个盘子烘干的时间倍数。

盘子的数量	$15n^2$	45n	$15n^2+45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1500	450	1950
100	150 000	4500	154 500
1000	15 000 000	45 000	15 045 000
10 000	1 500 000 000	450 000	1 500 450 000
100 000	150 000 000 000	4 500 000	150 004 500 000
1 000 000	15 000 000 000 000	45 000 000	15 000 045 000 000
10 000 000	1 500 000 000 000 000	450 000 000	1 500 000 450 000 000

图 2.1 增长函数的项之间的比较

这种记法称为 O()或大 O 记法。与问题大小无关、执行时间恒定的增长函数称为具有 O(1)的复杂度。通常，在确定增长函数和效率时，我们只关注程序或算法的可运行语句。

关键概念

算法的阶次是忽略该算法的增长函数中的常量和其他次要项，只保留主项而得出的。

例如，不管问题是大是小，运行赋值语句和 if 语句一次，其复杂度就为 O(1)。因此，无论有多少这种语句被罗列在一起，其复杂度都为 O(1)。循环语句和方法调用语句可能会导致更高阶次的增长函数，因为运行这些语句时，可能会导致某个语句或语句系列（根据该问题的大小）将运行多次。我们将在本章后面的章节分别介绍它们。图 2.2 显示了一些增长函数及其渐进复杂度。

增 长 函 数	阶 次	标 记
$t(n)=17$	O(1)	常量型
$t(n)=3\log n$	O(log n)	对数型
$t(n)=20n-4$	O(n)	线性
$t(n)=12n\log n+100n$	O(n log n)	n log n
$t(n)=3n^2+5n-2$	$O(n^2)$	平方型
$t(n)=8n^3+3n^2$	$O(n^3)$	立方型
$t(n)=2^n+18n^2+3n$	$O(2^n)$	指数型

图 2.2 一些增长函数及其渐进复杂度

更正式地，增长函数 $t(n)=15n^2+45n$ 的复杂度是 $O(n^2)$，这意味着存在一个常量 m 和 $n(n_0)$的某个值，使得对于 $n>n_0$ 有 $t(n)\leqslant m\times n^2$。另一种表示方式是，算法的阶次为增长函数提供了一个上界。还需要注意的是，还有其他的相关表示符，如Ω表示的是函数的下界，而Θ表示的是函数的上下界。我们将主要讨论函数的阶次。

关键概念

算法的阶次为增长函数提供了一个上界。

由于函数的阶次是主要因素，所以其他项和常数通常甚至都不用考虑。所有具有相同阶次的算法，从运行效率的角度来说都认为是等价的。例如，如果两个算法完成相同任务具有不同的增长函数，比如，都是 $O(n^2)$，那么从运行效率上来说，两者大体上是相等的。

2.3　增长函数的比较

有人可能以为，随着处理器速度的提高以及大量廉价存储器的涌现，将不再需要算法分析了。但事实并非如此。处理器速度和存储器并不能弥补算法效率的差异。注意，在前面的讨论中，当讨论算法的阶次时，我们把常量作为无关紧要的项而忽略不计了。处理器速度的提高只会给增长函数添加常量。如果可能，寻找一个更高效的算法是一种比使用更快处理器的更好解决方法。

Aho、Hopcroft 和 Ullman（1974）提出了另外一种考察算法复杂度的方法。假设某个系统在给定时间内能完成 n 大小的问题，如果把处理器的速度提高 10 倍情况会是怎样？如图 2.3 所示，渐进复杂度为线性的情况比较简单。具有线性时间复杂度 n 的算法 A，其运行效率确实提升了 10 倍。然而，时间复杂度为 n^2 的 B 算法的运行效率则仅仅提升了 3.16 倍。为什么 B 算法的运行效率没有提高 10 倍呢？因为算法 B 的复杂度是 n^2，其运行效率提高了 10 的平方根倍，即 3.16 倍。

算法	时间复杂度	提速前能处理的最大问题	提速后能处理的最大问题
A	n	s_1	$10\ s_1$
B	n^2	s_2	$3.16\ s_2$
C	n^3	s_3	$2.15\ s_3$
D	2^n	s_4	$s_4+3.3$

图 2.3　处理器提速 10 倍后能处理的问题大小的增加倍数

关键概念

如果算法的运行效率低，那么从长远来说，使用更快的处理器也无济于事。

同样，对于复杂度为 n^3 的算法 C，其运行效率只提高 2.15，即 10 的立方根。对于具有指数复杂度的算法 D，这种情况最糟糕。其运行效率只提高 $\log_2 n$（这里是 3.3）。在要处理的问题巨大的情况下，处理器的提速没有什么帮助。

图 2.4 用图形方式显示了 n 值相对较小的情况下的各种增长函数。请注意，当 n 很小时，各算法差别很小。这就是说，如果能保证问题很小（5 或者更小），那么使用哪种算法真的是无关紧要。但是，如图 2.5 所示，当 n 变得很大时，增长函数之间的差距也就明显起来。

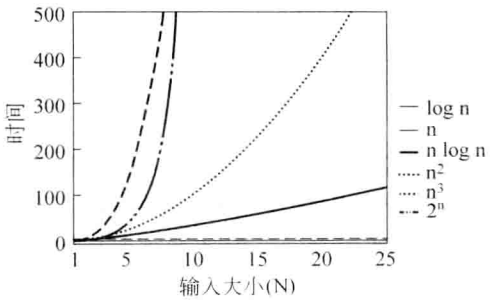


图 2.4 n 相对较小时，各种增长函数的比较

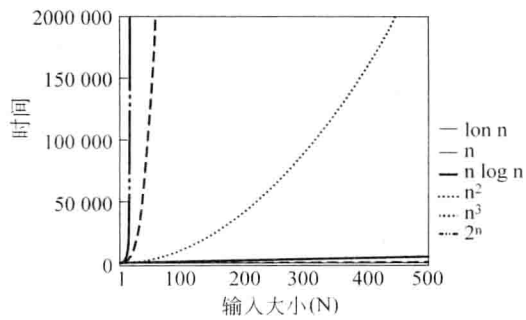


图 2.5 n 很大时，各种增长函数的比较

2.4 时间复杂度分析

2.4.1 循环运行的复杂度分析

要确定某个算法的阶次，常常需要确定某个特定语句或某个语句集运行的次数。因此，我们常常需要确定某个循环体的运行次数。要分析循环运行，首先要确定该循环体的阶次 n，然后用该循环要运行的次数乘以它。请记住，n 表示的是问题的大小。

关键概念

要分析算法的复杂度，通常需要分析循环的运行。

假设某个循环体的复杂度为 O(1)，那么下面循环的时间复杂度为 O(n)：

```
for  (int count = 0; count < n;  count++)
{
     /*复杂度为 O(1)的步骤系列*/
}
```

这是因为该循环体具有 O(1)的复杂度，且该循环结构将其运行 n 次。通常，如果某个循环结构以线性方式运行 n 次，且该循环体的复杂度为 O(1)，那么该循环的复杂度就为 O(n)。即使该循环设计为会跳过某些数量部分，只要所跳过的部分是线性的，该循环的复杂度仍是 O(n)。例如，如果上面的循环每次跳过一个数，那么该循环的增长函数为 n/2，但是因为常数不影响渐进复杂度，因此其阶次仍旧是 O(n)。

如果该循环的复杂度是对数级的，如下所示：

```
count = 1
while (count  <  n)
{
   count *= 2
   /*复杂度为 O(1)的步骤系列*/
}
```

则该循环是 O(logn)的。请注意，在算法复杂度中使用对数时，基本上是指以 2 为底的对数。其准确的写法是 $O(\log_2 n)$。因为每运行一次循环，count 值就被乘以 2，循环执行的次数为 $\log_2 n$。

关键概念

循环的时间复杂度等于循环体的复杂度乘以该循环运行的次数。

2.4.2　嵌套循环的复杂度分析

另一种更有趣的情景是循环出现嵌套时。在这种情况下，要计算出所求循环的复杂度，需用内层循环的复杂度乘以外层循环的复杂度。例如，下面嵌套循环：

```
for(int count = 0; count < n;  count++)
{
    for(int count2 = 0; count2 < n;  count2++)
    {
        /*复杂度为 O(1)的步骤系列*/
    }
}
```

其复杂度为 $O(n^2)$。内层循环体的复杂度为 O(1)，并且该内层循环将运行 n 次。这意味着内层循环的复杂度为 O(n)。把这个复杂度与外层循环运行的次数（即 n 次）相乘，就可得出嵌套循环的复杂度为 $O(n^2)$。

关键概念

分析嵌套循环的复杂度时，必须将内层循环和外层循环都考虑进来。

下面这个嵌套循环的复杂度是多少？

```
for  (int count = 0; count < n;  count++)
{
    for(int count2 = 0; count2 < n;  count2++)
    {
        /*复杂度为 O(1)的步骤系列*/
    }
}
```

在这种情况下，内层循环索引被初始化为外层循环索引的当前值。外层循环运行了 n 次。内层循环第 1 次被执行 n 次，第 2 次被执行 n–1 次，等等。但是，记住，我们只对主项感兴趣，而忽视常数项或任何其他次要项。如果复杂度是线性的，则不管要经过多少个元素，其阶次仍旧是 O(n)。因此，上面该代码的复杂度为 $O(n^2)$。

2.4.3 方法调用的复杂度分析

假设有如下代码段：

```
for (int count = 0; count < n; count ++)
{
   printsum(count);
}
```

从前面的讨论中可以知道，循环的阶次等于循环体的阶次乘以该循环运行的次数。但是，在上面情况下，循环体是一个方法调用。因此，在确定该代码段的阶次之前，必须首先确定这个方法的阶次。假设该方法的作用是，每调用一次时，就打印出 1～n 的所有整数之和，可以使用一个蛮力方法：

```
public void printsum(int count)
{
   int sum = 0;
   for (int I = 1; I < count; I++)
      sum += I ;
   System.out.println (sum);
}
```

printsum 方法的时间复杂度是多少呢？记住，只有可运行的语句才会增加时间复杂度，因此，在上面情况下，除了该循环之外，所有可运行语句的复杂度都为 O(1)。而该循环的复杂度为 O(n)，因此 printsum 方法的复杂度也就为 O(n)。现在来计算调用该方法的初始循环的时间复杂度，我们只需把 printsum 方法的复杂度（也就是循环体的复杂度）乘以该循环的运行次数即可。于是，调用了上面实现的 printsum 方法的复杂度为 $O(n^2)$。

从前面的讨论中可以知道，并不需要使用循环来计算 1～n 之和。事实上，我们知道有 $\Sigma^n_1 i=n(n+1)/2$。下面重新编写 printsum 方法，再看看时间复杂度会有些什么变化：

```
public void printsum(int count)
{
   int sum = 0;
   sum = count * (count + 1)/2;
   System.out.println (sum);
}
```

此时，printsum 方法的时间复杂度由一个赋值语句（其复杂度为 O(1)）和一个打印语句（其复杂度也为 O(1)）构成。于是，printsum 方法的时间复杂度现在就成了 O(1)。这就意味着调用该方法的循环的复杂度从 $O(n^2)$变成了 O(n)。从前面的讨论以及从图 2.5 中可知，这是一个很大的提高。从这里可以再一次看出简单算法与高效算法之间的差别。

如果一个方法体是由多个方法调用和循环构成的，情况又会是咋样呢？请看下面代码，它使用了前面的 printsum 方法：

```
public void sample (int n)
```

```
{
    printsum(n);                                    /*该方法调用的复杂度为 O(1)*/
    for (int count = 0; count < n; count ++)        /*该循环的复杂度为 O(n)*/
        printsum(count);
    for (int count = 0; count < n; count ++)        /*该循环的复杂度为 O(n2)*/
        for (int count2 = 0; count2 < n; count2 ++)
            System.out.println(count, cout2);
}
```

第一次调用 printsum 方法的复杂度为 O(1)，因为该方法的复杂度就为 O(1)。含有 printsum 方法调用的 for 循环的复杂度为 O(n)，因为该方法的复杂度为 O(1)，但该循环运行了 n 次。嵌套循环的复杂度为 $O(n^2)$，因为外层循环每运行一次，内层循环就要运行 n 次，而外层循环总共要运行 n 次。于是，整个方法的复杂度就是 $O(n^2)$。

sample 方法更为正式的表示形式是：

$$f(n) = 1+n+n^2$$

忽略常量和非主要项，就可以得出其时间复杂度为 $O(n^2)$。

在分析方法调用的时间复杂度时，需要处理的另一个问题是递归，在这种情况下，方法调用的是它自身，这将留待第 7 章讨论。

关键概念

- 软件必须有效地使用诸如 CPU 时间和内存之类的资源。
- 算法分析是计算机科学的基础。
- 增长函数表示与该问题大小相对应的时间或空间的使用。
- 算法的阶次是忽略该算法的增长函数中的常量和其他次要项，只保留主项而得出的。
- 算法的阶次为增长函数提供了一个上界。
- 如果算法的运行效率低，那么从长远来说，使用更快的处理器也无济于事。
- 要分析算法的复杂度，通常需要分析循环的运行。
- 循环的时间复杂度等于循环体的复杂度乘以该循环运行的次数。
- 分析嵌套循环的复杂度时，必须将内层循环和外层循环都考虑进来。

主要术语

analysis of algorithms：**算法分析**　计算机科学中主要关注软件算法效率的主题。

Big-Oh notation：**大 O 表示法**　用于表示函数的阶次、渐进复杂度的表示法。

growth function：**增长函数**　描述相当于问题大小的时间或空间使用的函数。

asymptotic complexity：**渐进复杂度**　增长函数的界限，由增长函数的主项确定的。渐进复杂度类似的函数，归为相同类型的函数。

自测题

SR 2.1 算法的增长函数与算法的阶次之间的区别是什么？

SR 2.2 对相关的工作量，为什么说提高 CPU 的速度并不一定能提高处理速度？

SR 2.3 如何使用算法的增长函数来确定其阶次？

SR 2.4 如何确定一个循环的时间复杂度？

SR 2.5 如何确定一个方法调用的时间复杂度？

练习题

EX 2.1 下列增长函数的阶次是多少？

a. $10n^2+100n+1000$

b. $10\,n^3-7$

c. $2^n+100\,n^3$

d. $n^2\log n$

EX 2.2 当 n=10，按升序排列上面练习题中的增长函数。当 n=1 000 000 时，将它们再次进行排序。

EX 2.3 编写必要的代码，以找出无序整数数组中的最大元素。该算法的时间复杂度是多少？

EX 2.4 请确定下面代码段的增长函数和阶次：

```
for (int count = 0; count < n; count ++)
    for (int count2 = 0; count2 < n; count2 =count2 + 2)
    {
        System.out.println(count, cout2);
    }
}
```

EX 2.5 请确定下面代码段的增长函数和阶次：

```
for (int count = 0; count < n; count ++)
    for (int count2 = 0; count2 < n; count2 =count2 * 2)
    {
        System.out.println(count, cout2);
    }
}
```

EX 2.6 图 2.1 显示了当 n 增长时，洗盘子示例的增长函数中的各个项之间的关系。请编写一个程序，为任一给定的增长函数创建这样一个表。

自测题答案

SRA 2.1 算法的增长函数表示了问题大小与解决方法的时间复杂度之间的确切关系。算法

的阶次表示了算法的渐进时间复杂度。随着问题的增大，算法的复杂度将不断接近该渐进复杂度。

SRA 2.2　当算法的阶次为常量（即 O(1)）或线性阶次（即 O(n)）时，提高 CPU 速度才会线性地提高处理速度。随着算法复杂度的增长，处理器速度的提高对复杂度的影响越来越小。

SRA 2.3　算法的阶次是忽略该算法的增长函数中的常量和其他次要项，只保留主项而得出的。

SRA 2.4　循环的时间复杂度等于循环体的复杂度乘以该循环运行的次数。

SRA 2.5　要计算方法调用的时间复杂度，先要计算出该方法自身的时间复杂度，然后把它代入到方法调用的时间复杂度计算中。

参考文献

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

第 3 章　集合概述——栈

本章学习目标

- 定义与集合相关的概念和术语
- 探讨 Java 集合 API 的基本结构
- 讨论集合的抽象设计
- 讨论集合的各种问题，如继承、多态性、泛型和接口
- 定义栈集合
- 使用栈集合解决问题
- 考察栈的数组实现

本章首先探讨集合以及用于实现集合的基本数据结构。本章定义了集合设计的相关问题和目标，这为集合的研究打下了基础。本章还将介绍一种称为栈的集合，并且以此为例展示与集合的设计、实现和使用等有关的问题。

3.1　集　　合

集合（collection）是一种聚集、组织了其他对象的对象。它定义了一种特定的方式，可以访问、管理所包含的对象（称为该集合的元素）。集合的使用者——通常是软件系统中的另一个类或对象——只能通过这些预定的方式与该集合进行交互。

多年以来，软件开发人员和研究人员定义了一些特定的集合类型。每种集合类型只致力于解决特定的问题类别。本书有很大篇幅都在探讨这些经典的集合。

关键概念

集合是一种聚集、组织了其他对象的对象。

集合可以分成两大类：线性集合和非线性集合。顾名思义，**线性集合**（linear collection）是一种其元素按直线方式组织的集合。而**非线性集合**（nonlinear collection）是一种其元素按某种非直线方式组织的集合，例如按层次组织或按网状组织。从这种意义上说，非线性集合也许根本就没有任何组织形式。

关键概念

集合中的元素通常是按照它们添加到集合的顺序，或者是按元素之间的某种内在关系来组织的。

图 3.1 给出了一个线性集合和一个非线性集合。线性集合中的元素是水平表示还是垂直表示通常是无关紧要的。

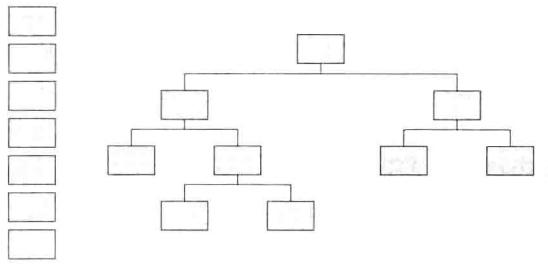


图 3.1　一个线性集合与一个非线性集合

集合中元素彼此之间的组织形式通常由下面情况之一决定的：

- 元素添加到集合的次序。
- 元素自身之间的一些内在关系。

例如，一种线性集合是在其末端添加新元素，因此，元素的次序由它们的添加次序确定的。另一种线性集合可以根据元素的一些特征保持有序。例如，人名清单可以根据姓氏、名字保持字母顺序。非线性集合元素的特定组织也可以按这两种方式来确定。

3.1.1　抽象数据类型

在某些时刻，**抽象**（abstraction）能隐藏某些细节。与一次性处理太多的细节相比，处理抽象会更容易一些。事实上，我们一天也离不开抽象。例如，如果需要操心构成汽车的零部件（如火花塞、活塞、传动等），那么我们就不可能再开车了。相反，我们只关心汽车的接口：方向盘、脚踏板以及一些其他控制装置。这些控制装置都是一种抽象，它们隐藏了底层的细节，使我们可以控制一个非常复杂的机器。

关键概念

集合是一种隐藏了实现细节的抽象。

与所有设计良好的对象一样，集合也是一种抽象。集合定义了一些接口操作，用户通过这些操作可以管理集合中的对象，例如添加、删除元素等。如图 3.2 所示，用户正是通过这个接口与集合交互的。但是，如何实现集合以满足定义则是另一个问题。实现了集合接口的类必须满足该集合的概念定义，但满足的方式可以有很多种。

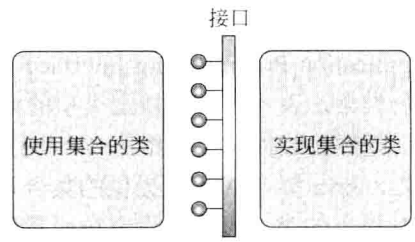


图 3.2　设计良好的接口隐藏了集合的实现

抽象是另一个重要的软件工程概念。在大型软件系统中，任何某一个人都不可能马上掌握该系统的所有细节。系统被划分成一些抽象的子系统，规定出这些子系统的目标以及它们之间的交互作用。然后，可以将这些子系统分派给不同的开发人员或开发小组；他们将开发出符合规范的

子系统。

对象是用于创建集合的一种完美机制，因为只要设计正确，对象的内部工作对系统其他部分而言是被**封装的**（encapsulated）。几乎在所有情况下，在类中定义的实例变量的可见性都应声明为私有的（private）。因此，只有该类的方法才可以访问和修改这些变量。用户与对象的唯一交互只能通过其公用方法。公用方法表示了对象所能提供的服务。

在探讨集合的过程中，我们总是会强调实现与接口分离的思路。因此，对将要考察的每种集合，我们都应该考虑如下问题：

- 集合在概念上是如何操作的？
- 如何正式地定义集合的接口？
- 集合能解决什么类型的问题？
- 能用哪些方式来实现集合？
- 各种实现的优点和代价是什么？

在继续下面的讨论之前，先让我们定义与集合有关的其他一些术语。**数据类型**（data type）是一组值及作用于这些数值上的各种操作。Java 语言中所定义的原始数据类型就是一些基本的数据类型。例如，**整数**（integer）数据类型定义了一组数值以及作用于这些数值上的操作（如加法、减法等）。

抽象数据类型（Abstract Data Type，ADT）是一种在程序设计语言中尚未定义其值和操作的数据类型。ADT 的抽象性体现在，ADT 必须对其实现细节进行定义，且这些对用户是不可见的。因此，集合是一种抽象数据类型。

数据结构（data structure）是一种用于实现集合的编程结构集。例如，集合可以用一种大小固定的结构（如数组）来实现。这些定义以及实现与接口分离的设计决策（即集合本身与实现集合的数据结构是分离的）的一个有趣产物是，我们可以（而且通常如此）最终利用线性数据结构（例如数组）来实现非线性集合（例如树）。

ADT 和数据结构这两个词曾经有过多种使用方式。这里进行精准定义是为了避免混淆，后面我们将不断使用它们。本书将考察各种数据以及如何使用它们来实现各种集合。

关键概念

数据结构是一种用于实现集合的基本编程结构。

3.1.2 Java 集合 API

Java 程序设计语言拥有一个用于支持软件开发的庞大类库。其中有部分类库是以 API（Application Programming Interface，应用程序编程接口）的形式组织的。Java 集合 API 是一个类集，表示了一些特定类型的集合，这些类的实现方式各不相同。

既然已经提供了一组集合，为什么还要学习如何设计、实现集合呢？这里有几个原因。首先，Java 集合 API 所提供的集合，可能只是我们所需集合的一个子集。其次，Java 集合 API 提供的类，可能并没有按照我们所希望的方式来实现这些集合。第三，可能最为重要的一点是，对软件开发的研究，要求我们深入理解集合设计以及用数据结构实现集合所涉

及的一些问题。

在探讨各种类型的集合时，还将考察 Java 集合 API 中的一些相应类。每次我们都会分析所开发的各种实现，并将它们与标准库中类的实现方式进行对比。

3.2 栈 集 合

关键概念

栈的元素是按后进先出(LIFO)的方法进行处理的，最后进入栈中的元素最先被移出。

下面是一个集合的例子。**栈**（stack）是一种线性集合，其元素的添加和删除都是在同一端进行的。我们称栈是按**后进先出**（Last in, first out，LIFO）的方式进行处理的。也就是说，放置在栈中的最后一个元素，将是第一个被移出的元素。换句话说，栈中的元素是以它们放置到栈中相反的顺序来移出的。事实上，在计算科学中，栈的基本使用就是用于颠倒顺序（如一个取消操作）。

栈的处理过程如图 3.3 所示。我们通常垂直地绘制栈，并把栈的末端称为栈的顶部，栈的元素就是从其顶部添加和删除的。

回忆一下前面的讨论，我们定义了抽象数据类型（ADT），给定了一个特定的操作集，这些操作可以管理存储在数据结构中的元素。我们总是想用这个概念来正式地为集合定义各种操作，使用它所提供的功能。这样，就可以明确地把集合的接口与用于创建该集合的任何特定实现技术分离开。

栈的操作如图 3.4 所示。按照栈的术语，可以把元素压入（push）到栈中，把元素从栈中弹出（pop）。还可以查看（peek）栈顶元素，检查它或在需要的时候使用它，但不用真的把它从集合中移出来。这些也是常用的操作，可以用来确定栈是否为空，如果不为空，它含有多少个元素。

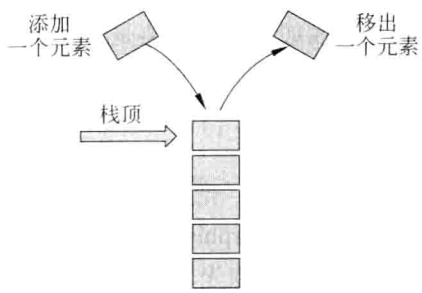


图 3.3　栈的示意图

操　作	描　述
push	添加一个元素到栈的顶部
pop	从栈的顶部移出一个元素
peek	查看栈顶部的元素
isempty	确定栈是否为空
size	确定栈的元素数目

图 3.4　栈的操作

关键概念

编程人员选择的数据结构，应与需要进行数据管理的类型相适应。

有时，对集合操作的命名习惯会有所变化。对栈而言，术语 push 和 pop 相对比较标准。但 peek 有时候又称为 top。

设计要点

在栈的设计中，可以看到栈的作用与使用栈的应用程序之间的分离。注意，如果 pop 或 peek 操作可以作用于空栈，那么栈的任何实现都要能抛出一个异常。集合的作用不是去确定如何处理这样一个异常，而只是把它报告给使用该栈的应用程序。同样，在栈中不存在满栈的概念。因此，应该由栈来管理它自己的存储空间，消除发生满栈的可能。

记住，集合的定义不是通用的。你可能会发现，在不同的书中，为特定数据结构定义的操作会有所不同。本书非常认真地定义了每种集合的操作，这样它们的作用就是固定的。

例如，在图 3.4 中，就没有这样一种操作，它能允许我们进入栈中去修改、删除或重新组织该栈中的元素。这是栈的一个本性——所有操作都是在一端进行的。如果明白了这点，那么，当求解某个问题时，如果需要访问集合中间或底部的元素，就不适合使用栈作为数据结构。

这里为栈提供了一个 toString 操作。这不是为栈定义的典型操作，并且有人认为，该操作会与栈预定义的行为发生冲突。但是，它提供了一个方便的途径，可以在不修改栈的情况下遍历和显示栈的内容，这对调试非常有用。

3.3 主要的面向对象概念

现在我们来看看在栈中将存储些什么。一种可能性是，每次需要时重新创建栈，以存储应用程序的特定对象类型。例如，如果需要一个字符串栈，就可以复制并粘贴栈代码，然后把对象的类型改为 String。从技术上来说，复制、粘贴和修改也是一种重用形式，但这种蛮力形式的重用并不是我们的目标。从最纯粹的形式来说，重用应该是，集合创建只进行一次编码，编译成字节码也只一次，然后就可以安全有效地处理要存储的任意对象。要实现这些目标，就必须考虑到**类型兼容性**（type compatibility）和**类型检查**（type checking）。类型兼容性是指把一个对象赋给引用的特定赋值是否合法。例如，下面赋值是不合法的，因为你不能把一个声明为 String 类型的引用赋给指向 Integer 类型的对象的指针：

```
String x = new Integer (10);
```

Java 语言提供了编译时类型检查，它可以标记出不合法的赋值。把数据存储在集合中的另一种可能性是，它可以利用**继承**（inheritance）和**多态**（polymorphism）的概念，以创建可以存储任意类型的对象的集合。要探讨这种可能性，我们必须先来探讨一下继承的概念。

3.3.1 继承与多态性

继承与多态性概念的完整介绍可参考附录 B。**多态引用**（polymorphic reference）是一个引用变量，它可以在不同地点引用不同类型的对象。继承可用于创建一个类层次，其中，一个引用变量可用于指向与之相关的任意对象。

从根本上说，一个 Object 引用可用于引用任意对象，因为所有类最终都是从 Object 类派生而来的。例如，ArrayList 使用多态性来存储一个 Object 引用。这也是 ArrayList 可用于存储任意类型的对象的原因所在。一个 ArrayList 在某个时候可以存储几种不同类型的对象，因为所有对象都是与 Object 兼容的。

从上面的讨论中你可能会认为，只需要在栈中存储 Object 引用，通过继承来创建一个可以存储任意类型的对象就行了。但是，这种解决方法可能会产生一些意想不到的后果。因为本章主要讨论的是用数组来实现栈，我们来看看使用多态引用和数组时会发生些什么。考虑图 3.5 所示的类。由于 Animal 是该图中所有其他类的超类，如下的赋值语句是允许的：

```
Animal creature = new Bird();
```

而下面赋值语句也是可以编译通过的：

```
Animal[] creatures = new Mammal[];
creatures[1] = new Reptile();
```

注意，根据定义，creatures[1]应该是 Mammal 或 Animal，但不是 Reptile。该代码可以编译，但在运行时会产生一个 java.lang.ArrayStoreException 异常。因此，由于使用 Object 类无法提供编译时类型检查，应寻找更好的解决方法。

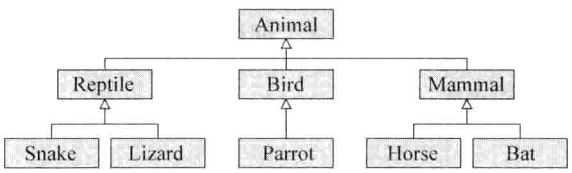


图 3.5　显示类层次的 UML 类图

3.3.2　泛型

从 Java 5.0 开始，Java 语言就允许用户基于**泛型**（generic type）来定义类了。也就是说，我们可以定义一个类，以便它能存储、操作和管理在实例化之前没有指定是何种类型的对象。泛型是我们进行集合讨论中的一个完整部分，本书的后面将介绍其实现。

假设需要定义一个名为 Box 的类，它可存储和管理其他对象。如前所述，利用多态性，只需把 Box 定义为存储指向 Object 类的引用。然后，任意类型的对象就都可以存储在 Box 中了。事实上，多种类型的不相干对象也的确都可以存储在 Box 中，但在代码中，丢失了一定程度的灵活性控制。

一种更好的方法是把 Box 类定义为可以存储泛型 T（可以使用任意所想的标识符，但使用 T 已经成为了一种习惯）。在类定义头的尖括号中包含指向该类型的引用。例如：

```
class Box<T>
}
    //用于管理类型 T 的对象的声明和代码
}
```

然后，当需要 Box 时，就用具体的类替换 T 来实例化它。例如，如果需要 Widget 对象的 Box，就可以如下进行声明：

```
Box<Widget> box1 = new Box<Widget>
```

变量 box1 的类型为 Box<Widget>。对 box1 对象而言，就是 Box 类用 Widget 替换了 T。现在假设需要一个 Box 来存储 Gadget 对象，可以进行如下声明：

```
Box<Gadget> box2 =new Box<Widget>
```

对 box2 来说，就是 Box 类用 Gadget 替换了 T。因此，尽管 box1 和 box2 都是 Box，但考虑到泛型因素，它们有不同的类型。这是一种更加安全的实现，因为此时的 box1 不能用来存储 Gadget（或其他），box2 也不能用来存储 Widget。泛型（如 T）不能被实例化。它只是一个占位符，允许我们去定义管理特定类型的对象的类，且只有当该类被实例化时，才创建该类型的对象。

至此，我们有了一种机制，使用泛型来创建可用于安全且高效地存储任意类型的对象的集合。下面继续栈的讨论。

3.4 使用栈计算后缀表达式

通常，算术表达式可写成中缀表示形式，即操作符位于操作数之间：

<操作数> <操作符> <操作数>

例如，表达式

```
4 + 5
```

在计算中缀表达式时，可依靠优先规则来确定操作符计算的顺序。例如，表达式：

```
4 + 5 * 2
```

的计算结果是 14 而不是 18，因为按照优先规则，在没有括号的情况下，乘法计算在加法之前。

在后缀表达式中，操作符在两个操作数之后。因此，后缀表达式的形式如下：

<操作数> <操作数> <操作符>

例如，后缀表达式

```
6 9 -
```

就等价于中缀表达式

```
6-9
```

通常，后缀表达式不比中缀表达式更容易计算，因为它不用考虑优先规则和括号。表

达式中的数值和操作符的顺序就足以确定计算结果。由于这个原因，程序设计语言编译器和运行时环境在其内部计算中往往使用后缀表达式。

后缀表达式的计算过程可以用一个简单的规则来表述：从左到右扫描，把每个操作符应用到其之前的两个紧邻操作数，并用该计算结果替代该操作符。最后，我们就剩下该表达式的最终值了。

请看前面的那个中缀表达式：

```
4 + 5 * 2
```

用后缀表示形式，该表达式可写成：

```
4 5 2 * +
```

下面使用计算规则来计算该表达式的最终值。我们从左开始扫描，直到遇到乘号操作符（*）。把该操作符应用到其之前的两个操作数（即 5 和 2），并用其计算结果（即 10）替换该操作符，于是就成了：

```
4 10 +
```

继续从左到右扫描，将立即遇到加号操作符（+），把该操作符应用到其之前的两个操作数（即 4 和 10），得到 14，这就是该表达式的最终值。

再看一个更复杂些的表达式。

```
(3 * 4 - (2 + 5)) * 4 / 2
```

其等价的后缀表达式为：

```
3 4 * 2 5 + - 4 * 2 /
```

应用计算规则的结果如下：

```
12  2 5 + - 4 * 2 /
12 7 - 4 * 2 /
5 4 * 2 /
20 2 /
10
```

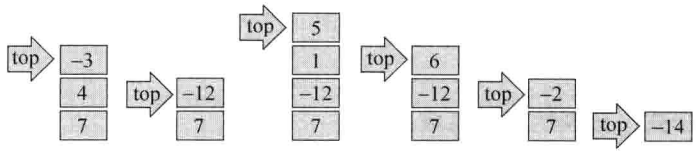
关键概念

栈是用于计算后缀表达式的理想数据结构。

下面考虑计算后缀表达式的程序设计。当遇到一个操作符时，计算规则应可以检索其之前的两个操作数。而且，大型的后缀表达式具有很多操作符和操作数需要管理。事实证明，栈是在这种情况下使用的一种很好集合。由栈提供的操作与后缀表达式计算过程有着很好的一致性。

使用栈来计算后缀表达式的算法描述如下：从左到右扫描表达式，依次标识出每个符号（操作符或操作数）。如果是操作数，则把它压入到栈中。如果是操作符，则从栈中

弹出两个元素，并把该操作符应用在这两个元素之上，然后把操作结果压入到栈中。当到达表达式的末尾时，栈中所剩余的元素就是该表达式的计算结果。如果试图从栈中弹出两个元素时，该栈中并没有，那么该后缀表达式是不正确的。同样，如果到达表达式末尾时，栈中剩余的元素不止一个，该表达式也是不正确的。图 3.6 描述了使用栈来计算后缀表达式。



7 4 -3 * 1 5 + / *

图 3.6 使用栈来计算后缀表达式

程序列表 3.1 可用于计算由用户输入的多个后缀表达式。它所使用的 PostfixEvaluator 类如程序列表 3.2 所示。

为使问题简单化，该程序假设表达式的操作数为整数，且为文字值（而不是变量）。当程序运行时，会反复接收和计算后缀表达式，直到用户选择退出为止。

程序列表 3.1

```
import java.util.Scanner;

/**
 * Demonstrates the use of a stack to evaluate postfix expressions.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixTester
{
    /**
     * Reads and evaluates multiple postfix expressions.
     */
    public static void main(String[] args)
    {
        String expression, again;
        int result;

        Scanner in = new Scanner(System.in);

        do
        {
            PostfixEvaluator evaluator = new PostfixEvaluator();
            System.out.println("Enter a valid post-fix expression one
```

```
                                token "+"at a time with a space between each
                                token (e.g. 5 4 + 3 2 1 - + *)");
            System.out.println("Each token must be an integer or
                                an operator (+,-,*,/)");
            expression = in.nextLine();

            result = evaluator.evaluate(expression);
            System.out.println();
            System.out.println("That expression equals " + result);

            System.out.print("Evaluate another expression [Y/N]? ");
            again = in.nextLine();
            System.out.println();
        }
        while (again.equalsIgnoreCase("y"));
    }
}
```

程序列表 3.2

```
import java.util.Stack;
import java.util.Scanner;

/**
 * Represents an integer evaluator of postfix expressions. Assumes
 * the operands are constants.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixEvaluator
{
    private final static char ADD = '+';
    private final static char SUBTRACT = '-';
    private final static char MULTIPLY = '*';
    private final static char DIVIDE = '/';

    private Stack<Integer> stack;

    /**
     * Sets up this evalutor by creating a new stack.
     */
    public PostfixEvaluator()
    {
```

```
        stack = new Stack<Integer>();
    }

    /**
     * Evaluates the specified postfix expression. If an operand is
     * encountered, it is pushed onto the stack. If an operator is
     * encountered, two operands are popped, the operation is
     * evaluated, and the result is pushed onto the stack.
     * @param expr string representation of a postfix expression
     * @return value of the given expression
     */
    public int evaluate(String expr)
    {
        int op1, op2, result = 0;
        String token;
        Scanner parser = new Scanner(expr);

        while (parser.hasNext())
        {
            token = parser.next();

            if (isOperator(token))
            {
                op2 = (stack.pop()).intValue();
                op1 = (stack.pop()).intValue();
                result = evaluateSingleOperator(token.charAt(0), op1, op2);
                stack.push(new Integer(result));
            }
            else
                stack.push(new Integer(Integer.parseInt(token)));
        }

        return result;
    }

    /**
     * Determines if the specified token is an operator.
     * @param token the token to be evaluated
     * @return true if token is operator
     */
    private boolean isOperator(String token)
    {
        return ( token.equals("+") || token.equals("-") ||
                 token.equals("*") || token.equals("/") );
```

```
    }

    /**
     * Peforms integer evaluation on a single expression consisting of
     * the specified operator and operands.
     * @param operation operation to be performed
     * @param op1 the first operand
     * @param op2 the second operand
     * @return value of the expression
     */
    private int evaluateSingleOperator(char operation, int op1, int op2)
    {
        int result = 0;

        switch (operation)
        {
            case ADD:
                result = op1 + op2;
                break;
            case SUBTRACT:
                result = op1 - op2;
                break;
            case MULTIPLY:
                result = op1 * op2;
                break;
            case DIVIDE:
                result = op1 / op2;
        }

        return result;
    }
}
```

PostfixEvaluator 类使用 java.util.Stack 类来创建栈。java.util.Stack 类是 Java 集合 API 提供的两个栈实现之一。在第 4 章的 Deque 接口中，将看到另一个实现。

evaluate 方法执行前面描述的计算算法，由 isOperator 和 evalSingleOp 方法支持。注意，在 evaluate 方法中，只把操作数压入到栈中。当遇到操作符时就使用它来进行计算，并不把它压入到栈中。这与我们所讨论的计算算法是一致的。操作数是作为一个 Integer 对象而不是作为一个 int 基本数值被压入到栈中的，这是因为栈被设计为存储对象的。

当遇到一个操作符时，最近压入到栈中的两个操作数被弹出。注意，第一个弹出的操作数其实是表达式的第二个操作数，而第二个弹出的操作数则是表达式的第一个操作数。对于加法和乘法，这种顺序无关紧要，但对减法和除法则是重要的。

同时还要注意，后缀表达式程序假定输入的后缀表达式是合法的，这意味着它包含的

是一个结构正确的操作符和操作数集。后缀表达式在下面情况下是不合法的：

（1）当遇到一个操作符时，栈中没有两个操作数；

（2）当表达式中的符号用完后，栈中的数值不止一个。这两种情况都表明，该表达式的格式有问题，通过在程序的恰当地方检查栈的状态，可以捕获这两种错误情况。

该程序最重要的方面大概要算是定义了栈的类的使用了。此时，我们并不知道栈是如何实现的。只需要相信该类能完成它自己的工作即可。本示例使用了 java.util.Stack 类，也可以使用任何实现了栈的类，只要它完成栈的预期操作即可。从计算后缀表达式的角度来说，栈实现的方式是无所谓的。图 3.7 显示了后缀表达式计算程序的一个 UML 类图。该图显示，PostfixEvaluator 类使用了 java.util.Stack 的一个 Integer 实例，并把 Integer 绑定为泛型 T。在 UML 图中并没有包含这个级别的细节内容。

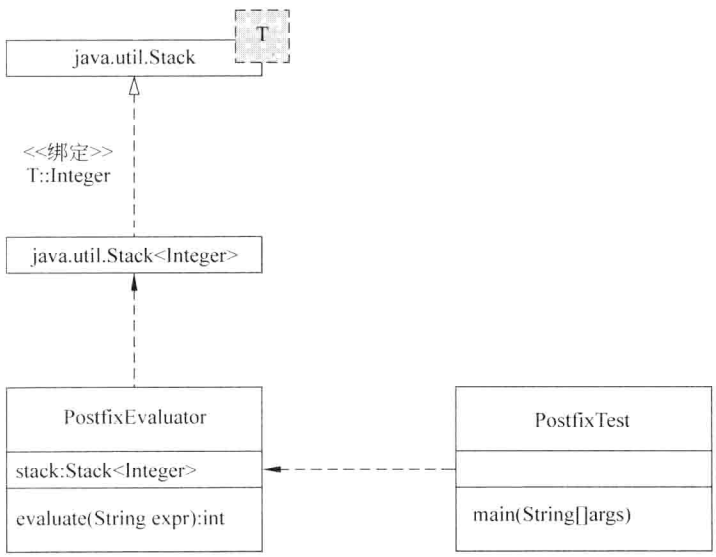


图 3.7 后缀表达式计算程序的 UML 类图

3.4.1 Javadoc

在继续之前，先介绍一下在程序列表 3.1 和 3.2 中的文档注释风格。这些是 Javadoc 注释，这些注释以这样一种格式编写：使得 javadoc 工具（JDK 工具集的一部分）可以解析该注释并能抽取出有关类和方法的信息。Javadoc 注释以/**开始，以*/结束。

Javadoc 可用于创建关于类集的在线 HTML 文档。你应该已经见过这种文档，在线的 Java API 文档就是用这种技术创建的。当对 API 类（及其注释）进行修改时，再次运行 javadoc 以生成该文档。这是一种非常聪明的方式，确保文档不会滞后代码的发展。

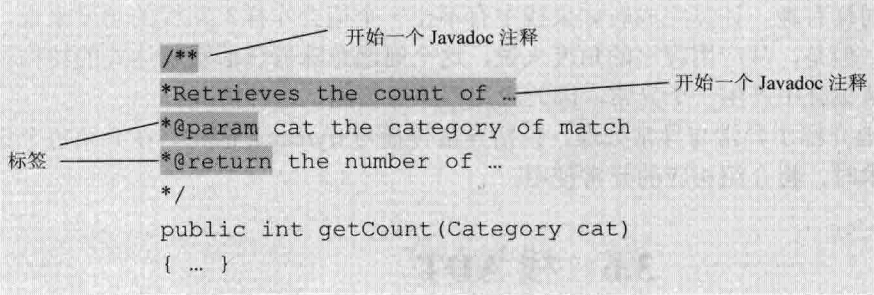
这不并不是 Java API 专属的。任何程序或类集的文档都可以使用 Javadoc 来生成。即使不用 Javadoc 来生成在线文档，你也应该使用 Javadoc 注释风格，因为它是为 Java 代码添加注释的官方标准。

Javadoc 标签用于标识特定类型的信息。例如，@author 标签用于标识编写代码的程序员。@version 标签用于指定代码的版本号。在方法的头部，@return 标签用于表明由该方

法返回的值，@param 标签用于标识传递给该方法的每个参数。

这里不再深入介绍 Javadoc，但本书将使用 Javadoc 注释风格。

一个方法的 Javadoc 注释




```
/**
*Retrieves the count of …
*@param cat the category of match
*@return the number of …
*/
public int getCount(Category cat)
{ … }
```

3.5 异　　常

介绍每个集合时，我们都要探讨一个概念，那就是异常行为。在异常情况下，集合应采取什么动作？有些情况是集合本身固有的。例如，对于栈，如果试图从空栈中弹出元素时，会发生什么？在这种情况下，使用何种数据结构来实现集合无关紧要，都会应用异常。在有些情况下，使用的是自己构建的数据结构来实现集合的。例如，当我们使用数组来实现栈时，如果试图压入一个元素到栈中但数组已满，会发生什么？让我们花费些时间来进一步讨论这个概念。

在 Java 程序中出现问题时，会产生异常或错误。**异常**（exception）就是一个对象，它定义了一种非正常或错误的情况。异常由程序或运行时环境抛出，可以按预期的被捕获或被正确处理。**错误**（error）与异常类似，只不过错误往往表示一种无法恢复的情况，且不必去捕获它。Java 有一个预定义的异常和错误集，当程序运行时它们可能会发生。

在前面的后缀计算表达式示例中，有几个潜在的异常情况。例如：

- 当要压入元素时栈已满。
- 当要弹出元素时栈为空。
- 计算完成时如果栈中有不止一个值。

请分别看看这些情况。当要压入元素时栈已满，这是数据结构而不是集合的问题。从概念上来说，并不存在满栈这回事。现在我们知道，这是不切实际的，所有数据结构最终都会达到一个极限。但是，即使是达到了物理极限，栈也不是满的，只是实现栈的数据结构为满的。下一节将讨论处理这种情况的策略。

要弹出元素时栈为空会如何？这是一种必须去处理该问题的异常情况，而不是数据结构。在前面的后缀计算示例中，如果试图从栈中弹出两个操作数但没有两个操作数时，说明该后缀表达式的结构不正确。在这种情况下，集合需要报告该异常，然后，应用程序必须解释该异常。

关键概念

错误和异常表示不正常或不合法的处理。

第三种情况同样有趣。计算完成时如果栈中有不止一个值会咋样？从栈的角度来说，这不是一个异常。但是，从应用程序的角度来说，这个问题意味着该后缀表达式的结构不正确。因为不会从集合中产生一个异常，因此应用程序必须测试这种情况。

附录 B 完整地介绍了异常与异常处理，包括异常传播与 try/catch 语句。在介绍每个集合的具体实现技术时，将介绍相应的异常使用。

3.6 栈 ADT

为了便于把接口操作与实现接口的方法分离开，可以为集合定义一个 Java 接口结构。Java 接口为所有集合定义操作集提供了一种正式的机制。

关键概念

Java 接口定义了一个抽象方法集，有助于把抽象数据类型的概念与其实现分隔开来。

回忆可知，Java 接口定义了一个抽象方法集，描述了每个方法的名称，但没有描述方法体。实现接口的类为该接口中的类提供了定义。接口名可以用作引用类型，它可以赋给实现了该接口的所有类的所有对象。

程序列表 3.3 为栈集合定义了一个 Java 接口。我们使用集合名后跟抽象数据类型（abstract data type）的缩写 ADT 来为集合接口命名。因此，StackADT.java 包含了栈集合的接口。它被定义为 jsjf 包的一部分，该包含有本书中所介绍的所有集合类和接口。

关键概念

通过使用接口名作为返回类型，方法就不会局限于实现栈的任何特定类。

注意，栈接口被定义为 StackADT<T>，表示作用于泛型 T。在该接口的方法中，各种参数和返回值的类型往往用泛型 T 来描述。当实现该接口时，将会用某种数据类型来替换泛型 T。

程序列表 3.3

```
package jsjf;

/**
 * Defines the interface to a stack collection.
 *
```

```
 * @author Lewis and Chase
 * @version 4.0
 */

public interface StackADT<T>
{
    /**
     * Adds the specified element to the top of this stack.
     * @param element element to be pushed onto the stack
     */
    public void push(T element);

    /**
     * Removes and returns the top element from this stack.
     * @return the element removed from the stack
     */
    public T pop();

    /**
     * Returns without removing the top element of this stack.
     * @return the element on top of the stack
     */
    public T peek();

    /**
     * Returns true if this stack contains no elements.
     * @return true if the stack is empty
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this stack.
     * @return the number of elements in the stack
     */
    public int size();

    /**
     * Returns a string representation of this stack.
     * @return a string representation of the stack
     */
    public String toString();
}
```

栈接口

```
                保留字          接口名
                  |               |
public interface StackADT <T> ———————— 泛型参数
{
  ...
}
```

本书每次介绍接口、类或系统时，都会给出该接口、类或系统的 UML 描述。这将有助于读者熟悉 UML 描述，并为其他类和系统创建 UML 描述。图 3.8 显示了 StackADT 接口的 UML 描述。

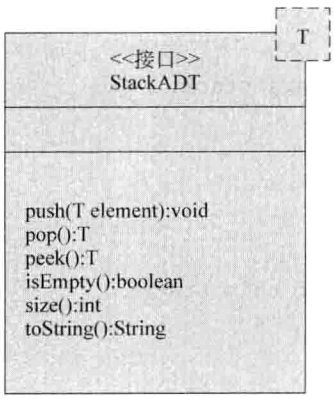


图 3.8 StackADT 接口的 UML 描述

在计算机科学中，栈的使用很频繁。例如，字处理器中的取消操作通常就是用栈来实现的。对某个文档进行修改（添加数据、删除数据、进行格式修改等）时，字处理器通过把每个操作的表示压入栈来跟踪这些操作。如果要取消一个操作，字处理器会从栈中弹出最近执行的操作并退回上一操作。如果要再次进行取消操作（即取消所执行的倒数第二个操作），会从栈中弹出另一个元素。在大多数字处理器中，很多操作都可以用这种方式来实现回退。

设计要点

取消操作通常是使用一种名为 drop-out 的栈来实现的。drop-out 栈的基本操作与一般栈的相同（即 push、pop 和 peek 等）。唯一的不同是，drop-out 栈对能存储的元素数量有限制，一旦达到限制，如果有新元素要压入，那么栈底的元素将被从栈中丢弃掉。有关 drop-out 栈的开发留作练习。

3.7 用数组实现栈

到目前为止，在栈集合的讨论中，我们描述了它的基本性质，以及允许用户与之进行交互的各种操作。就软件工程术语而言，可以说已完成了对栈集合的分析。我们还使用了一个栈来求解一个特定的问题，但并不知道实现的具体细节。现在让我们把注意力转到实现细节上来。实现表示栈的类有多种方式。如前所述，Java 集合 API 提供了多种实现方式，包括 Stack 类和 Deque 接口。本节将考察一种使用数组来存储栈中的对象的实现策略。在第 4 章中将考察实现栈的另一种技术。

关键概念

集合操作的实现不应该影响到用户与该集合进行交互的方式。

为了探讨这种实现，必须回忆一下 Java 数组的几个主要特性。存储在数组中的元素是按 0～n－1 进行索引的，其中 n 为数组的单元总数。一个数组就是一个对象，它与数组所存储的对象分别被实例化。当我们说到对象的数组时，其实说的是对象引用的数组，如图 3.9 所示。

记得区分集合与实现它的基本数据结构之间的区别。我们的目标是设计一个高效的实现，提供在抽象数据结构栈中定义的每种操作的功能。数组只是一种用来存储对象的便利数据结构。

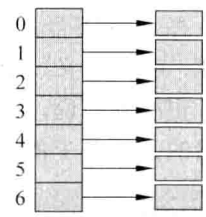


图 3.9　对象引用的数组

3.7.1　管理容量

当创建一个数组对象时，会为它分配一定数量的单元数，其元素可以存储在其中。例如，下面的实例化语句创建了一个可存储 500 个元素的数组，其索引号为 0～499：

```
Object[] collection = Object[500];
```

数组中的单元数量称为该数组的**容量**（capacity）。该值存储在数组的 length 常量中。数组一旦创建好，其容量是不能改变的。

当使用数组来实现集合时，必须处理该数组的所有单元都已存储了元素的情况。也就是说，由于使用的是固定大小的数据结构，在某些时候，该数据结构可能会“满”。但是，只是因为该数据结构满了，就意味着集合也满了吗？

集合设计中的一个关键问题是，当新元素加入到一个满的数据结构中时应该做些什么。这里存在三种基本选择：

- 实现往集合添加一个元素的操作，如果集合为满，抛出一个异常。
- 实现 add 操作，该操作返回一个可由用户检查的状态标识符，以便查看该 add 操作是否成功。
- 当需要时可扩展集合的容量，这样，该集合就永远也不会满了。

关键概念

如何处理异常情况，决定了集合或集合的用户能否控制某个行为。

在前两个步骤中，集合的使用者必须明白，该集合可能为满，需要时应采取处理步骤。对于这些解决方法，还可以提供另外的操作，允许用户去检查集合是否为满，并可按需扩展该集合的容量。这些解决方法的好处是，给予用户更多的集合容量控制。

然而，由于我们的目标是要把接口与实现分离开，因此第三种选择更有吸引力。通常，集合的容量属于实现细节，对用户是不可见的。而且，容量问题是这种实现方法特有的。用于实现集合的其他技术，如我们将在下一章中介绍的，不会受限于固定的容量，因此不用去处理这个问题。

在本书所介绍的解决方法中，我们更倾向于通过自动扩展集合的容量来实现固定大小的集合。在程序设计项目中，其他选择也偶尔会提及。

3.8 ArrayStack 类

在 Java 集合 API 框架中，类名表示了基本数据结构和集合，本书也遵循这种命名习惯。因此，我们定义一个名为 ArrayStack 的类来表示一个基于数组实现的栈。

更准确地说，我们定义一个名为 ArrayStack<T>的类来表示一个基于数组实现的栈，该存储的是泛型 T 的对象。当实例化一个 ArrayStack 对象时，将指定泛型 T 所表示的内容。

关键概念

出于运行效率的考虑，基于数组的栈实现总是使栈底位于数组的索引 0 处。

栈的数组实现可以通过以下 4 个假设来设计：

（1）该数组是一个对象引用的数组（其数据类型在栈实例化时确定）；

（2）栈底总是在数组的索引 0 处；

（3）栈的元素是按顺序并连续地存储在数组中；

（4）有一个整数变量 top，该变量保存了紧跟栈顶元素后的数组索引号。

图 3.10 显示了当前栈含有元素 A、B、C 和 D 的情况，假设这些元素是按顺序压入到栈中的。为了简化这个图，这些元素显示在该数组而不是引用该数组的对象中。注意，变量 top 既表示了下一个单元（将要压入元素就存储在这里），也表示了当前栈中的元素数量。

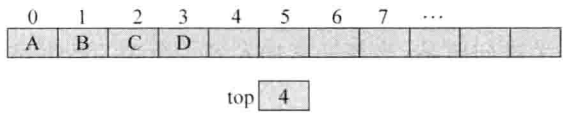


图 3.10 栈的数组实现

在这个实现中，栈底总是存储在数组的索引 0 处，栈在更大的索引处增大和缩小。如果栈在数组中顺序是颠倒的，那么效率就会更高。如果栈顶在索引 0 处，则需要考虑进行处理。

从这些假设我们可以确定，ArrayStack 类需要一个常量来存储默认的容量，需要一个变量来跟踪栈顶，以及一个变量来存储该栈。下面是该类的头。注意，ArrayStack 类是 jsjf 包的一部分，使用了名为 jsjf.exceptions 的包。

```
package jsjf;

import jsjf.exceptions.*;
import java.util.Arrays;

/**
* An array implementation of a stack in which the bottom of the
* stack is fixed at index 0
```

```
*
* @author Lewis and Chase
* @version 4.0
*/
public class ArrayStack<T> implements StackADT<T>
{
    private final int DEFAULT_CAPACITY = 100;

    private int top;
    private T[] stack;
```

3.8.1　构造函数

ArrayStack 类有两个构造函数，一个使用的是默认容量，另一个使用的是指定容量。

```
/* 使用默认容量创建一个空栈*/
public ArrayStack()
{
    top = 0;
    stack = (T[]) (new Object[DEFAULT_CAPACITY]);
}

/*使用指定容量创建一个空栈,参数 initialCapacity 表示的是指定容量*/
public ArrayStack (int initialCapacity)
{
    top = 0;
    stack = (T[]) (new Object[initialCapacity]);
}
```

回忆一下可知，这是方法重载的一个极好示例（即具有相同名称的两个方法，其区别仅是参数列表不同）。

从前面有关泛型的讨论中可以知道，不能实例化一个泛型对象。这意味着不能实例化任何泛型数组。这就是为什么在两个构造函数中都有这样有趣的一行：

```
stack = (T[]) (new Object[DEFAULT_CAPACITY]);
```

注意，该行实例化了一个 Object 数组，然后把它转换为一个泛型数组。这里将产生一个未检验类型转换的编译时警告，应该是 Java 编译器不能保证这种类型转换的安全性。从前面的讨论中我们已经知道，为了获得灵活性和泛型的安全性，产生这种警告也是值得的。在该语句之前使用如下 Java 注释，可以禁止这种警告：

```
@SuppressWarnings("uncheched")
```

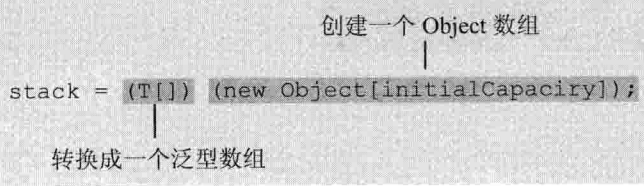
常见错误

对不熟悉泛型的程序员来说，常见的一个错误是试图创建一个泛型数组：

```
stack = new T[initialCapaciry];
```

泛型（包括泛型数组）不能被实例化。这就是为什么我们必须创建一个存储 Object 引用的数组，然后把它转换成泛型数组的原因。

创建一个泛型数组



3.8.2 push 操作

要把一个元素压入到栈中，只需把它插入到数组中由变量 top 指定的下一个可用位置中即可。但是，在这样做之前，必须确定该数组是否已满，是否需要扩充其容量。存储该元素后，还必须更新变量 top 的值，以便它能继续表示栈的元素数目。

下面是实现这些步骤的代码：

```
/**
* Adds the specified element to the top of this stack, expanding
* the capacity of the array if nessary.
* @param element generic element to be pushed onto stack
*/

public void push (T element)
{
   if (size() == stack.length)
      expandCapacity();
   stack[top] = element;
   top++;
}
```

当需要时，实现 expandCapacity 方法来使数组的大小加倍。当然，由于数组一旦实例化以后就不能改变其大小，因此该方法是创建一个更大的数组，并把旧数组的内容复制到新数组中。它是类的一个支持方法，因此可以实现为私有的。

```
/**
* Create a new array to store the contents of this stack with
* twice the capacity of the old one.
*/

private void expandCapacity()
```

```
{
    stack = Arrays.copyOf(stack, stack.length * 2);
}
```

图 3.11 显示了往图 3.10 中压入元素 E 的结果。

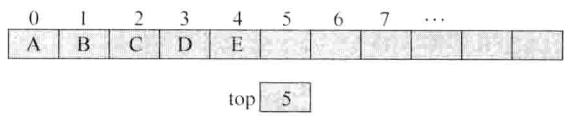


图 3.11　压入元素 E 后的栈

对于栈的数组实现，其 push 操作由以下步骤构成：

- 确保该数组不是满的。
- 把数组的 top 引用设置为要加入到栈中的对象。
- 增加 top 的值。

以上每一步的复杂度都为 O(1)，因此，该操作的复杂度也为 O(1)。我们可能想知道 expandCapacity 方法的时间复杂度，以及它对 push 方法的影响。该方法的确含有一个 for 循环，我们可能会认为其复杂度为 O(n)。但是，相对于 push 方法调用的次数来说，调用 expandCapacity 方法的次数很少，因此可以在所有 push 方法中分摊 expandCapacity 方法的复杂度。

3.8.3　pop 操作

pop 操作移出并返回栈顶元素。对于数组实现，这意味着返回的是索引 top–1 的元素。但是，在试图返回一个元素之前，必须确保栈中至少有一个要返回的元素。

基于数组的 pop 操作可以如下实现：

```
/**
* Remove the element at the top of this stack and return a
* reference to it.
* @return element removed from top of stack
* @throws EmptyCollectionException if stack is empty
*/
public T pop() throws EmptyCollectionException
{
   if (isEmpty())
       throw new EmptyCollectionException("Stack");
    top--;
    T result = stack[top];
    stack[top] = null;

    return result;
}
```

当调用 pop 方法时，如果栈为空，会抛出一个 EmptyCollectionException 异常。否则，top 的值被减少，存储在该位置的元素，被存储在一个临时变量中，这样就可以返回该值。然后把数组中的该单元设置为空。图 3.12 显示了 pop 操作作用于图 3.11 中的栈之后的结果，这将使得该栈回到以前的状态（与图 3.10 相同）。

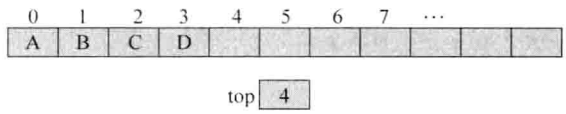


图 3.12 弹出栈顶元素后的栈

数组实现的 pop 操作由以下几个步骤构成：

- 确保栈不为空。
- 减小 top 计数器。
- 设置一个临时引用等于 stack[top]的元素。
- 设置 stack[top]为空。
- 返回该临时引用。

所有这些步骤的复杂度都为 O(1)。这样，数组实现的 pop 操作具有的复杂度为 O(1)。

3.8.4 peek 操作

peek 操作返回一个指向栈顶元素的引用，但不把它从数组移出。对于数组实现，这意味着将返回一个指向 top–1 处的元素。这一步的复杂度为 O(1)，因此 peek 操作的复杂度也是 O(1)。

```
/**
* Returns a reference to the element at the top of this stack.
* The element is not removed from the stack.
* @return elememt on top of stack
* @throws EmptyCollectionException if stack is empty
*/
public T peek() throws EmptyCollectionException
{
   if (isEmpty())
        throw new EmptyCollectionException("Stack");

    return stack[top-1]
}
```

3.8.5 其他操作

isEmpty、size 和 toString 操作及其分析分别留作程序设计项目练习。

3.8.6　EmptyCollectionException 类

至此，我们已经查看了 ArrayStack 类的实现，这里再来看看异常处理。当底层的数据结构满了时，就需要使用集合来处理这种情况，因为这是集合内部的一个问题。另一方面，当集合为空时，如果 pop 或 peek 操作试图访问集合中的一个元素，也需要抛出一个异常。这种情况揭示了这种集合的使用问题，而不是集合本身。

在 Java 中，异常是一个类，因此可以使用 Java API 中提供的已有异常，或自己创建。此时，可以选择创建一个特定的空栈异常。然而，创建一个参数化的异常，就可以在其他任何集合中重复使用这个异常。程序列表 3.4 显示了 EmptyCollectionException 类。注意，这个异常类是从 RuntimeException 类扩展而来的，然后利用 super 引用，使用了其父类的构造函数。

程序列表 3.4

```
package jsjf.exceptions;

/**
 * Represents the situation in which a collection is empty.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class EmptyCollectionException extends RuntimeException
{
    /**
     * Sets up this exception with an appropriate message.
     * @param collection the name of the collection
     */
    public EmptyCollectionException(String collection)
    {
        super("The " + collection + " is empty.");
    }
}
```

3.8.7　其他实现

本章考查了栈的概念，使用 Java API 中的 Stack 类来解决后缀计算问题，然后实现了我们自己的 ArrayStack 类，该类使用数组来存储栈的基本元素。

我们还没有完成这个栈的实现。第 4 章将考查另一种技术，即使用链式结构而不是数组来实现集合，并实现一个 LinkedStack 类。

等有了这两种实现技术（基于数组和基于链表），我们就可以探讨其他更多的集合了。

关键概念

- 集合是一种聚集、组织了其他对象的对象。
- 集合中的元素通常是按照它们添加到集合的顺序，或者是按元素之间的某种内在关系来组织的。
- 集合是一种隐藏了实现细节的抽象。
- 数据结构是一种用于实现集合的基本编程结构。
- 栈的元素是按后进先出（LIFO）的方法进行处理的，最后进入栈中的元素最先被移出。
- 编程人员选择的数据结构，应与需要进行数据管理的类型相适应。
- 栈是用于计算后缀表达式的理想数据结构。
- 错误和异常表示的是不正常或不合法的处理。
- Java 接口定义了一个抽象方法集，有助于把抽象数据类型的概念与其实现分隔开来。
- 通过使用接口名作为返回类型，方法就不会局限于实现栈的任何特定类。
- 编程人员必须认真考虑如何处理异常，以及在哪一层处理异常。
- 集合操作的实现不应该影响到用户与该集合进行交互的方式。
- 如何处理异常情况，决定了集合或集合的用户能否控制某个行为。
- 出于运行效率的考虑，基于数组的栈实现总是使栈底位于数组的索引 0 处。

主要术语

abstraction：抽象 通过隐藏或忽略某些细节，使概念更容易管理的方法。

abstract data type：抽象数据类型 一种数据类型，其值和操作在编程语言中尚未定义。

class hierarchy：类层次 通过继承创建的类之间的关系，其中，某个类的子类又可以成为其他类的父类。

collection：集合 是一种聚集、组织了其他对象的对象。

data structure：数据结构

（1）对象的一种组织形式，使得某些操作的执行更高效。

（2）一种用于实现集合的编程结构。

exception：异常 一个对象，它定义了一种非正常或错误的情况。

generic type：泛型 对象类型的占位符，在引用它的类被实例化之前，其类型是不确定的。

inheritance：继承 一种面向对象原则，就是从已有类派生出一个新类的过程。

interface：接口

（1）两个事物相互作用的一种方式。

（2）一个公用方法的集合，使得一个对象可以与另一个对象相互作用。

Java Collection API：Java 集合 API Java API 的一个子集，用于表示或处理集合。

LIFO

（1）后进先出（Last in，first out）。

（2）对集合的一种描述，其中，最后添加的元素，将是第一个删除的元素。

polymorphism：多态　一种面向对象原则，使得一个引用可以多次指向相关但不同的对象类型，且其中调用的方法是在运行时与代码绑定的。

pop：弹出　栈的一种操作，把栈顶的元素删除。

push：压入　栈的一种操作，把元素添加到栈顶。

stack：栈　一种线性集合，其元素的添加和删除都是在同一端以 LIFO 方式进行的。

自测题

SR 3.1　何谓集合？

SR 3.2　何谓数据类型？

SR 3.3　何谓抽象数据结构？

SR 3.4　何谓数据结构？

SR 3.5　何谓抽象？它有哪些优点？

SR 3.6　为什么说类是抽象数据类型的极佳表示？

SR 3.7　栈有哪些特征行为？

SR 3.8　栈的 5 种基本操作是什么？

SR 3.9　还可以为栈实现其他哪些操作？

SR 3.10　请给出继承的定义。

SR 3.11　请给出多态性的定义。

SR 3.12　设有图 3.5 所示的示例，请列出 Mammal 的子类。

SR 3.13　设有图 3.5 所示的示例，下面代码能编译吗？

```
Animal creature = new Parrot();
```

SR 3.14　设有图 3.5 所示的示例，下面代码能编译吗？

```
Horse creature = new Mammal();
```

SR 3.15　Java 语言中的泛型有什么作用？

SR 3.16　后缀表示法有何优点？

练习题

EX 3.1　比较并对比一下数据类型、抽象数据类型和数据结构。

EX 3.2　请列举出 Java 集合 API 中的集合，并标记出在本书中介绍过的集合。

EX 3.3　请定义抽象的概念，并解释一下，为什么在软件开发中抽象是很重要的。

EX 3.4　通过以下操作来跟踪栈 X：

```
X.push(new Integer(4));
X.push(new Integer(3));
Integer Y = X.pop();
X.push(new Integer(7));
```

```
X.push(new Integer(2));
X.push(new Integer(5));
X.push(new Integer(9));
Integer Y = X.pop();
X.push(new Integer(3));
X.push(new Integer(9));
```

EX 3.5 设有来自上一练习题的结果栈 X，经过下面每个操作后的结果如何？

a. Y = X.peek();

b. Y = X.pop();
 Z = X.peek();

c. Y = X.pop();
 Z = X.peek();

EX 3.6 isEmpty、size 和 toString 方法的时间复杂度应如何？

EX 3.7 请说明字处理器中的取消操作是如何通过栈来实现的。请给出特定示例，并描绘出采取各种动作后栈的内容。

EX 3.8 在后缀表达式计算示例中，当遇到一个操作符时，会弹出两个最近的操作数，这样就可以计算子表达式。在子表达式中，把所弹出的第一个操作数看作是第二个操作数，而第二个弹出的操作数看作是第一个操作数。给出并解释一个描述这种解决方法的重要性的示例。

EX 3.9 有 5 个整数(12, 23, 1, 45, 9)，如何使用栈把这些元素的顺序颠倒为(9, 45, 1 23, 12)。

EX 3.10 请解释，如果栈顶位于索引 0 处，那么用数组实现的栈的算法和时间复杂度会是啥样？

程序设计项目

PP 3.1 请完成本章的 ArrayStack 类的实现。特别是完成 peek、isEmpty、size 和 toString 方法的实现。

PP 3.2 设计和实现一个应用程序，读取从用户输入的一个句子，并反向显示每个词的字符。使用栈来颠倒每个词的字符。

PP 3.3 修改后缀表达式计算问题的解决方法，以便在用户输入表达式时能检查其合法性。当遇到错误情况时，能发出恰当的错误消息。

PP 3.4 在本章的数组实现中，变量 top 总是指向实际栈顶的下一个数组位置。重写这个数组实现，使得 stack[top]是实际的栈顶。

PP 3.5 有一种名为 drop-out 栈的数据结构，它与其他栈一样，只不过，如果栈的大小为 n，当压入 n+1 个元素时，将丢弃第一个元素。请使用数组来实现 drop-out 栈（提示：用环形数组实现会更有意义）。

PP 3.6 请用三个栈来实现一个整数加法器。

PP 3.7 请用栈来实现一个从中缀到后缀的转换器。

PP 3.8 请实现一个名为 reverse 的类，它使用栈把由用户输入的元素集按反序输出。

PP 3.9　请创建一个图形应用程序，该程序提供一个用于压入栈和弹出栈的按钮，一个用于接收要压入栈的字符串的文本域，以及另一个用于显示栈（每个操作完成后）的内容的文本域。

自测题答案

SRA 3.1　集合是一个对象，它可以聚集和组织其他对象。

SRA 3.2　数据类型是程序设计语言中的一组值及作用于这些数值上的各种操作。

SRA 3.3　抽象数据类型是一种在程序设计语言中尚未定义其值和操作的数据类型，它必须由编程人员来定义。

SRA 3.4　数据结构是一种用于实现抽象数据类型的对象集。

SRA 3.5　抽象是这样一种概念，它隐藏了操作和数据存储的基本实现，以便简化集合的使用。

SRA 3.6　本质上，类就提供了抽象，因为只有为其他类提供服务的那些方法才是公用可见的。

SRA 3.7　栈是一种后进先出（LIFO）的结构。

SRA 3.8　栈的 5 种基本操作是：

push——添加一个元素到栈的一端。

pop——从栈的前端移出一个元素。

peek——返回一个指向栈的前端元素的引用。

isEmpty——如果栈为空返回真，否则返回假。

size——返回栈的元素数目。

SRA 3.9　makeEmpty()、destroy()、full()。

SRA 3.10　继承就是从已有类派生出一个新类的过程。新类自动地含有初始类的部分和全部变量和方法。因此，如需定制类，编程人员可以给派生类添加新的变量和方法，或者修改继承来的变量和方法。

SRA 3.11　术语“多态性”可定义为“具有多种形式”。多态引用是一个引用变量，它可以在不同地方指向不同类型的对象。通过多态引用调用的某个方法，在每次调用时都可以发生变化。

SRA 3.12　Mammal 的子类是 Horse 和 Bat。

SRA 3.13　可以。因为父类和任何超类的引用变量，存有指向其子孙类的引用。

SRA 3.14　不可以。子类的引用变量不能存有指向其父类和超类的引用。要进行这种赋值，必须显式地把父类转换为子类，如下所示：

```
(Horse creature = (Horse) (new Mammal());
```

SRA 3.15　从 Java 5.0 开始，允许用户基于泛型来定义一个类。也就是说，我们可以定义一个类，它存储、操作和管理的对象的数据类型直到该类实例化时才被指定。这使得结构的创建不仅可以操作“泛型”元素，而且还可以提供类型检查。

SRA 3.16　后缀表示法不需要优先规则，而在计算中缀表达式时是需要优先规则的。

第 4 章　链式结构——栈

本章学习目标

- 描述如何用引用来创建链式结构
- 比较链式结构与基于数组的结构
- 探讨链表管理技术
- 讨论利用单独结点对象来构成链表结构的必要性
- 利用链表实现栈集合

本章讨论了一种创建数据结构的技术，利用引用来创建对象之间的链接。链式结构是软件开发的基础，尤其对集合的设计与实现更是如此。与数组实现解决方案比较，链式结构既有优点，也有缺点。

4.1　链接作为引用

第 3 章讨论了集合的概念，并具体讨论了一种集合——栈。我们定义了栈集合的操作，并且利用基于数组的基本数据结构设计了一种栈的实现，本章将探讨一种完全不同的数据结构设计方式。

关键概念

对象引用变量可以用来创建链式结构。

链式结构（linked structure）是一种数据结构，它使用对象引用变量来创建对象之间的链接。链式结构是基于数组的集合实现的主要替代方案。在讨论链式结构的各种问题之后，我们将定义一种新的栈集合实现，它使用了一种基本的链式数据结构。然后还演示了它的使用。

回忆可知，对象引用变量存放的是对象的地址，表示该对象在内存中的存储位置。如下声明创建了一个名为 obj 的变量，该变量只够存放对象的数字地址：

```
Object obj;
```

通常，对象引用变量存放的具体地址是无关紧要的。也就是说，重要的是要能够利用该引用变量来访问对象，对象具体存储在内存的哪里并不重要。因此，我们通常并不是显示地址，而是将引用变量描绘成一种“指向”对象的名字，如图 4.1 所示。在这种上下文中用到的引用变量有时又称为**指针**（pointer）。



图 4.1　指向另一个对象的对象引用变量

请看这样一种情况：某个类将指向该类的另一对象的引用定义为实例数据。例如，假定有个名为 Person 的类，它含有某个人的姓名、地址和其他相关信息。现在假设 Person 类除了含有这些数据之外，还包含着一个指向另一 Person 对象的引用变量。

```
public class Person
{
        private String name;
        private String address;
        private Person next;                    //到另一个 Person 对象的链接
        //其他内容
}
```

只需用这个类就可以创建一个链式结构。一个 Person 对象含有指向第 2 个 Person 对象的链接。第 2 个 Person 对象又含有指向第 3 个 Person 对象的引用，而第 3 个 Person 对象又含有指向第 4 个 Person 对象的引用，依此类推。这种类型的对象有时又称为**自引用的**（self-referential）。

这种关系就构成了链表的基础。链表是一种链式结构，其中的一个对象可以指向另一个对象，从而在链表中创建一个对象的线性次序，如图 4.2 所示。在链表中存储的对象通常泛称为该链表的**结点**（node）。

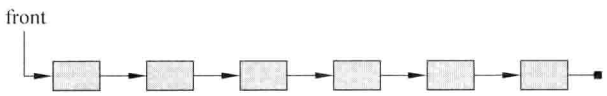


图 4.2　一个链表

注意，需要一个单独的引用变量来表示链表的首结点。链表终止于其 next 引用为空的结点。

关键概念

链表由一些对象构成，其中每个对象指向了链表中的下一对象。

链表只是链式结构的其中一种。如果建立的类含有多个指向对象的引用，就可以创建更复杂的链式结构，如图 4.3 中描绘的那种链式结构。链接的管理方式表明了这种链式结构的特定组织形式。

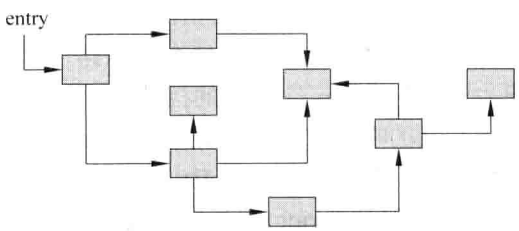


图 4.3　一种复杂的链式结构

下面将关注链表的细节内容，这些链表技术很多同样也适用于更复杂的链式结构。

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的PDF图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ是：**89039855**，大家可以在 QQ上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。